A Performance Test Design Method and its Implementation Patterns for Multi-Services Systems

vorgelegt von
Diplom-Ingenieur
George Din

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr. Ing.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Anja Feldmann
Berichter: Prof. Dr. -Ing. Ina Schieferdecker
Berichter: Prof. Dr. phil.-nat. Jens Grabowski

Tag der wissenschaftlichen Aussprache: 8.09.2008

Berlin 2009
D 83

Abstract

Over the last few years, the scope of telecommunication services has increased dramatically, making network infrastructure-related services a very competitive market. Additionally, the traditional telecoms are now using Internet technology to provide a larger range of services. The obvious outcome, is the increase in the number of subscribers and services demanded.

Due to this complexity, the performance testing of continuously evolving telecommunication services has become a real challenge. More efficient and more powerful testing solutions are needed. This ability depends highly on the workload design and on the efficient use of hardware resources for test execution.

The performance testing of telecommunication services raises an interesting problem: how to create adequate workloads to test the performance of such systems. Traditional workload characterization methods, based on requests/second, are not appropriate since they do not use proper models for traffic composition. In these environments, users interact with the network through consecutive requests, called transactions. Several transactions create a dialog. A user may demand in parallel two or more services and different behavioural patterns can be observed for different groups of users.

This thesis proposes a performance testing methodology which copes with the afore mentioned characteristics. The methodology consists of a set of methods and patterns to realize adequate workloads for multi-service systems. The effectiveness of this methodology is demonstrated throughout a case study on IP Multimedia Subsystem performance testing.

Zusammenfassung

In den letzten Jahren hat sich das Angebot an Telekommunikationsdiensten erweitert, was dazu geführt hat, dass der Markt für Dienste, die sich auf Netzwerkinfrastrukturen beziehen, mittlerweile sehr umkämpft ist. Ausserdem werden die traditionellen Telekommunikationssysteme mit Internet Technologien kombiniert, um eine grössere Auswahl an Diensten anbieten zu können. Daraus resultieren offensichtlich eine Zunahme der Teilnehmeranzahl und ein erhöhter Dienst-Bedarf.

Infolge dieser Komplexität werden Leistungstests der sich kontinuierlich entwickelnden Telekommunikationsdienste zu einer echten Herausforderung. Effizientere und leistungsfähigere Testlösungen werden benötigt. Das Leistungsvermögen hängt ab vom Workload Design und von der effizienten Nutzung der Hardware für die Testdurchführung.

Die Leistungstests der Telekommunikationsdienste führen zu einer interessanten Problemstellung: Wie soll man adäquate Lastprofile erstellen, um die Leistung solcher Systeme zu testen? Traditionelle Methoden zur Darstellung der Last, die auf Anfrage/Sekunde basieren, sind nicht zweckmässig, da sie keine geeigneten Modelle zur Anordnung des Datenverkehrs nutzen. In diesen Umgebungen soll der Nutzer mit dem Netzwerk über fortlaufende Anfragen, sogenannten Transaktionen, interagieren. Mehrere Transaktionen erzeugen einen Dialog. Ein Benutzer kann gleichzeitig zwei oder mehrere Dienste abrufen und es können verschieden Navigationsmuster für verschiedene Benutzergruppen beobachtet werden.

Diese Promotion schlägt eine Methodologie für Leistungstests vor, die sich mit den vorher genannten Charakteristika beschäftigt. Diese Methodologie setzt sich aus Verfahrensweisen und Modellen zusammen, die eine adäquate Last von Multi-Dienst Systemen realisieren sollen. Die Leistungsfähigkeit dieser Methodologie wird in einer Fallstudie nachgewiesen, die sich mit Leistungstests von IMS-Systemen (IP Multimedia Subsystem) befasst.

Acknowledgements

The work in this thesis required a large effort on my part, but this effort would not have been possible without the support of many people. My most special thanks are for Bianca for her love and patience with my endless working days. I warmly thank my parents for their love, education and accepting my living away.

I especially thank my advisers Professor Dr. Ina Schieferdecker and Professor Dr. Jens Grabowski. I wrote a thesis related to TTCN-3 and have been coordinated by the creators of this language, I cannot imagine ever topping that. I thank Ina Schieferdecker for giving me the opportunity to work in TTmex and IMS Benchmarking projects and for supporting my work and my ideas over the years. I also thank her for the many discussions we had and for guiding me over the years to achieve always the best results. Her knowledge, suggestions and numerous reviews contributed much to the results and the form of this thesis. I thank Jens Graboswki for the valuable suggestions and for insufflating me a high level of quality and professionalism.

I would like to express my gratitude to Professor Dr. Radu Popescu Zeletin for providing me with excellent working conditions during my stay at Fraunhofer FOKUS and for giving me the advice to start a career on testing. My sincere thanks are also due to Professor Dr. Valentin Cristea and Ivonne Nicolescu for advicing me during my studentship to follow an academical direction.

I also thank the members of the TIP research group at FOKUS (later MOTION-Testing group) for many fruitful discussions and for providing the environment in both technical and non-technical sense that made this work not just possible but even enjoyable. Sincere thanks to my colleagues Diana Vega and Razvan Petre for helping me during the implementation of the software. Also thank Diana Vega and Justyna Zander-Nowicka for the tireless efforts of proof-reading the document. I am grateful to Zhen Ru Dai (aka Lulu), Justyna Zander-Nowicka and Axel Rennoch who didn't let me forget that live is more than writing a PhD thesis.

I consider myself fortunate to be involved in two technical challenging projects. I greately enjoyed working with Theofanis Vassiliou Gioles, Stephan Pietch, Dimitrios Apostolidis and Valentin Zaharescu during the TTmex project. Many thanks I owe to the colleagues in the IMS Benchmarking project: Tony Roug, Neal Oliver, Olivier Jacques, Dragos Vingarzan, Andreas Hoffmann, Luc Provoost and Patrice Buriez for their efforts, ideas and debates about IMS benchmarking that sharpened my arguments. Also especial thanks to INTEL for providing me with the latest hardware technology for doing the experimental work. These experiences were the true definition of getting a doctorate to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The worthwhile problems are the ones you can really solve or help solve, the ones you can really contribute something to.
– Richard Feynman

The service providers (SPs) are evolving their networks from legacy technologies to "fourth generation" technologies which involves: (1) evolution of "traditional" wire-line telecoms standards to Voice over IP (VoIP) standards [DPB+06], (2) evolution of Global System for Mobile Communications (GSM) and Code Division Multiple Access (CDMA) networks to 3rd Generation Partnership Project (3GPP) [3GP08] standards, e.g., Universal Mobile Telecommunications System (UMTS) [KR07], (3) introduction of Wireless Local Area Network (WLAN) standards, e.g., IEEE 802.16 [oEI08], for both data and voice communications [RBAS04]. The current direction is to realise a convergence point of these trends into a set of technologies termed the IP Multimedia Subsystem (IMS). The concept behind IMS is to support a rich set of services available to end users on either wireless or wired User Equipments (UEs), provided via a uniform interface. Services are provided via an "overlay" technique over multiple service provider networks.

The introduction of many new telecoms services and technologies is making it difficult to satisfy service quality requirements [Afu04], [TSMW06]. The growing number of users adds additional performance requirements to the upcoming telecommunication technologies. Therefore, the telecommunications service provider's survival depends on its ability to prepare for changes in customer needs, as well as changes in regulation and technology [Jeu99].

Due to this complexity, the performance testing of continuously evolving telecommunication services has become a real challenge. More efficient and more powerful testing solutions are needed. These test solutions require a good performance test design and an efficient use of hardware resources for performance test execution. There is an urgent need to do research for a performance testing methodology that utilises the characteristics of the interactions between users and services, and methods to create adequate performance tests which simulate realistic traffic patterns [AKLW02]. This can only be achieved by addressing the overall performance test design, test architecture and test execution aspects common to various performance testing scenarios.

The survey in [WV00] indicates that very little research has been published in the area of performance testing. The traditional methods to design performance tests do not use adequate traffic models to simulate the real behaviour of a user population. They are based on statistical observations of the traffic, e.g., packets/second, and try to reproduce a similar traffic which satisfies the observed statistics [SK74], [ÁCV02]. This is unfortunately not enough to simulate characteris-

tics of a realistic workload as they do not take into account the behavioural paths of a user, the probability of a user to create a call etc. as required for testing of telecommunication services.

This thesis elaborates a set of methods and patterns to design and implement efficient performance tests. The methodology is designed for telecoms systems which offer a large number of telecommunication services and typically have to handle a large number of requests in short periods of time. Furthermore, the telecommunication services are deployed in a network which typically has to serve a large number of users.

Different from the previous approaches, the creation of the performance tests is realised by instantiating a user behaviour model for each simulated user. The performance test relates to four parts: workload, performance metrics, performance test plan and performance test report. The *workload* describes the traffic of a large number of individual simulated users, each performing an individual scenario where a scenario is the set of actions to use a service. The rate at which scenarios are attempted in the performance test is controlled by a *traffic-time profile* defined for the test. A traffic-time profile is so designed that the rate of scenario attempts remains constant for sufficient time to collect a statistically significant data set. As long as any user can execute calls for any service, a *traffic set* concept is used to assign users to services. The traffic set also describes the proportions of services composition. For example, it describes how along the test, a service S1 is instantiated 70% while a service S2 is instantiated 30%.

1.1 Scope of the Thesis

This thesis presents a methodology that defines processes for effectively designing and implementing performance tests for multi-service systems. It presents a set of methods to create workloads, performance metrics and performance test procedures. Additionally, it discusses how these performance tests can be technically realised.

The *construction of efficient performance tests*, in terms of resource consumption, requires the development of efficient methods and algorithms at various levels in the design of the performance test framework. An efficient performance test design starts with the selection of the most appropriate conception of the workload. There are many possibilities to architect a workload, but the concern is to find the one which fits better on a certain hardware- and operating system configuration. Furthermore, the execution model of the workload is fundamental due to two major aspects. Firstly, the workload's abstract elements, e.g., state machines, parallel processes and user data, are mapped to underlying operating system elements like threads, memory, network protocol units. Secondly, the workload parts must be implemented in such ways that the separation and the distribution over several test nodes are possible. Both aspects must be considered when designing a performance test.

The problem this thesis addresses can be stated as follows:

> *How can performance tests for multi-service systems be designed and realised efficiently?*

Before having a deeper insight into the performance testing methodology elements, the following questions should be examined more closely:

1. What are the performance characteristics of the tested system? The target of performance testing is to evaluate the performance of a system. This is achieved by identifying the important system characteristics related to performance and by evaluating them. Some examples of general performance characteristics are: volume, throughput and latency. The performance characteristics are evaluated by collecting data in the form of performance metrics such as: maximal number of users, maximal load intensity or transaction delays.

2. Which are the parameters to control a performance test? A number of performance test parameters should be defined in order to control the performance test. They typically regard workload aspects such as number of users, number of calls per second, test duration but also mathematical properties of the traffic such as the probability of a scenario to happen along a test run.

3. How does a user behaviour relate to a workload? A workload is created as a composition of the behaviours of a huge number of users. A user is an instance of a user behavioural model that defines the number and rate at which an individual user makes scenario attempts. Each user may be selected during test execution and be assigned to a particular scenario.

4. How can workloads be efficiently implemented? Workloads are descriptions of the interactions between a test system (TS) and the system under test (SUT). These descriptions are, often, informal and depict the message flows and the performance test procedure, e.g., how to increase the load, how many users to create. The workloads are implemented in a programming language which binds the abstract concepts like user, transaction etc., to platform specific elements: threads, data buffers or network channels. In this respect, several implementation patterns [DTS06] for mapping workload abstract elements to platform specific elements can be identified.

In order to realise the proposed performance testing methodology, the thesis deliberates the following aspects:

Performance test information model. A performance test is constructed according to a performance test information model. The information model identifies a limited number of concepts that essentially characterise the performance test. Examples of such concepts are: test scenario, performance metric, performance test parameter or performance test procedure. These elements are instantiated for each performance test. The information model is based on the concept that any workload presented to an SUT starts with the behaviour of an individual user. When a user interacts with the SUT, he/she does so with a particular goal, e.g., to make a call. The SUT may provide a variety of ways to accomplish this goal. However, the high-frequency actions are relatively limited in number, and can be captured by a manageable set of scripts. Individual users may differ in the relative speed with which the actions are performed, but this behaviour can be described by a probability model.

Performance test execution. Today's technologies provide powerful scripting engines which permit writing tests in high level languages. However, more overhead and resources demand on the test servers come along with the "ease-of-use". A test execution model considers all aspects of mapping abstract elements such as test case, message, communication channel to Operating System (OS) elements such as process, memory, network. Several design patterns are identified and discussed in relation with test parallelisation to increase test efficiency.

Using the Testing and Test Control Notation, version 3 (TTCN-3). The performance tests are experimentally realised by use of TTCN-3 language as test specification and implementation lan-

guage. TTCN-3 is suitable for specifying performance tests and its language elements simplify the design of complex workloads. However, the concrete execution and distribution of the executable tests is out of consideration of TTCN-3 specification. Therefore, an additional test specification element is needed to describe the test configuration on a target network of test nodes consisting potentially of one test node only.

IMS Benchmarking case study. The work presented in this thesis originates from the author's participation in the IMS Benchmarking project [FOK07], a joint project between Fraunhofer FOKUS [Fok08] and Intel [Int08]. The project aimed at the specification, execution and validation of performance benchmarks for IMS enabling solutions. The continuation of this work resulted in the participation in the IMS Benchmark Specification [TIS07] work item at the European Telecommunications Standards Institute (ETSI). Consequently, most concepts presented in this work have been applied and evaluated within this case study.

1.2 Structure of the Thesis

This section presents an overview of the chapters of this thesis. Figure 1.1 highlights the relation between their contents.

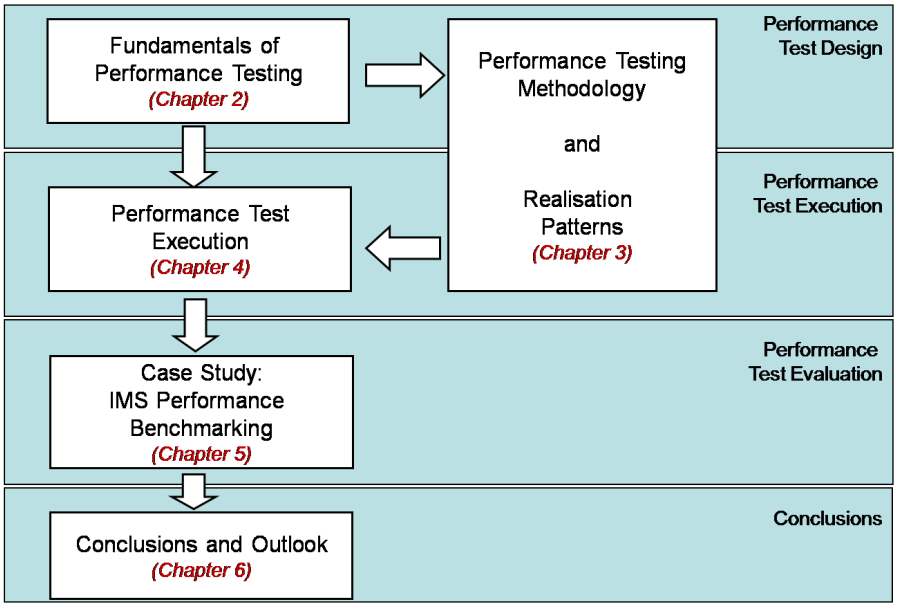

Figure 1.1: Chapters Overview

- *Chapter 2: Fundamentals of Performance Testing* - this chapter gives an overview on established foundations introduced and employed along the thesis. This includes a characterisation of performance testing in general and reviews performance testing of multi-service systems.
- *Chapter 3: Performance Testing Methodology and Realisation Patterns* - this chapter introduces the developed methodology for performance testing. Based on the proposed design requirements, workload and test execution design patterns are derived.

- *Chapter 4: Performance Test Execution* - this chapter provides design guidelines for implementing a performance testing platform. The guidelines are based on the experience of implementing a TTCN-3 based distributed execution platform.
- *Chapter 5: Case Study: IMS Performance Benchmarking* - this chapter presents the IMS benchmarking case study which serves as the basis for experimental work. The requirements for performance testing an IMS platform and applying the methodology to realise performance tests are discussed. The prototype implementation uses most of the workload patterns discussed in Chapter 3.
- *Chapter 6: Conclusions and Outlook* - the concluding remarks and discussion of future works complete the thesis.

1.3 Dependencies of Chapters

Chapters 3 to 5 use concepts presented in Chapter 2. Nevertheless, a reader who is familiar with the concepts related to performance testing may skip to Chapter 3.

Chapter 3 describes the developed performance test design methodology which is applied in the case study presented in Chapter 5. It is important to understand the concepts discussed in Chapters 4. Chapter 5 presents the case study on IMS performance testing.

A reader who just wants to get a quick overview on the topics of this thesis should read the summaries which are provided at the end of each chapter and the overall summary and conclusions given in Chapter 6.

Chapter 2

Fundamentals of Performance Testing

Nothing is particularly hard if you divide it into small jobs.
– Henry Ford

Performance testing is a complex conceptual and technical activity for answering questions related to the performance of a system. [1]. Some of the typical questions [Bar04a] that performance testing answers are: Does the application respond quickly enough for the intended users? Will the application handle the expected user load and more? How many users are possible before it gets "slower" or stops working? Performance testing requires not only concepts, methods and tools but also a broad understanding of software and systems. This chapter reviews concepts related to performance testing, which are referred to throughout the thesis.

2.1 Concepts of Performance Testing

In a general view, *performance testing* is a qualitative and quantitative evaluation of a SUT under realistic conditions to check whether *performance requirements* are satisfied or not. As a quality control task, the performance testing must be performed on any system before delivering it to customers. On a testing road map, performance testing comes at the end of the test plan, after functional testing including conformance, integration and interoperability takes place. However, the evaluation of performance is particularly important in early development stages, when important architectural choices are made [DPE04].

Binder [Bin99] defines performance testing in relation with performance requirements. Performance tests are designed to validate performance requirements which are expressed either as time intervals in which the SUT must accomplish a given task, as performance throughput, e.g., the number of successful transactions per unit of time, or as resource utilisation. Performance testing finds out how responsive the tested system is, how fast it fulfils its tasks under various load conditions but also discovers problems like the degradation of performance with time or increased loading, software errors or malfunctions that are sensitive to loading and also thresholds at which overloading might occur.

Gao et al. [GTW03] define performance testing as the activity to validate the system performance and measure the system capacity. There are three major goals defined. The first goal is to validate

[1] The notion of system is used here in a broader sense: it can mean an application, a network, a part of a network, etc.

the system ability to satisfy the performance requirements. A second goal is to find information about the capacity and the boundary limits. This information helps customers to compare different solutions and select the one which fits best in terms of costs and performance requirements. The last goal is to assist the system designers and developers in finding performance issues, bottlenecks and/or to further tune and improve the performance of the system.

The performance testing activities act upon a target system that usually, in the testing area, is called SUT. The SUT can be every element of a complex system: a) hardware parts together with their controlling drivers, e.g., hard-disk, CPU [DRCO05] or b) network elements running embedded applications, e.g., routers, sensor controllers [SB04] or c) applications running on a node, e.g., Web servers, Web services, databases [MA98], [LF06], [ZGLB06], or d) a whole network with distributed network services running on them, e.g., telecommunication infrastructure, distributed applications [BHS+99], [SSTD05], [TSMW06].

The focus of this thesis is the performance testing of systems from the last two categories. Those systems are reactive systems with respect to the interaction with users and interaction between parts of the system. They are supposed to deliver multiple services to a large number of users. For instance, the telecommunication network infrastructure should be able to sustain several million users [HMR02], [CPFS07]. The SUT is considered as a whole and the test system interacts with it over well-defined interfaces and protocols. These systems provide many services which are accessed by users during a communication session. The user requests require responses from the server side. A typical session between the user and the server consists of a sequence of request/response pairs where all messages in a session are characterised by the same session identifier. The sessions can vary from one pair of request/response, e.g., simple Web based applications up to several pairs, e.g., call establishment in a UMTS network [KR07]. The simplest example for such SUT is a Web server where each request is answered with a response - this interaction is also known as content-delivery interaction. However, in telecommunication services every service consists of sequences of several interactions.

2.1.1 Performance Testing Process

A *performance testing process* consists of all engineering steps starting from requirements to design, implement, execute and analyse the results of a performance test. A road map to elaborate a performance test process is described in [GTW03]. This process or parts of it is applied in many case studies [WV00], [Men02a].

The process starts with the selection of *performance requirements*. In this step, the performance testing engineer has to identify which features characterise the performance of the tested system. High number of transactions per unit of time, fast response times, low error rate under load conditions are examples of very common performance requirements. However, there are also other specific requirements, which might be considered for a system in particular: availability, scalability or resources utilisation. A detailed view on performance requirements selection is provided in [Nix00]. This paper presents a performance requirements framework, which integrates and catalogues performance knowledge and the development process.

The next step is to define the *workload*. In [GTW03] the term *performance evaluation models* is used instead of workload. The workload comprehends the performance testing focus points, test interfaces and scenarios of interest. The test engineer has to identify for each scenario the sequence of actions the test system should interchange with the SUT, interaction protocols and input data sets. The workload includes also *traffic models* which describe traffic patterns for the

request arrival rates [GJK99] with varying inter-arrival times and varying packet length [SSR97]. A model often used in practise for the description of traffic patterns is that of Markov Modulated Poisson Processes [FMH93].

Based on the identified scenarios, the *performance metrics* need to be defined. A classification of the types of metrics can also be found in [Jai91]. Metrics can be: throughput metrics, e.g., number of transactions per unit of time; resource utilisation, e.g., CPU consumption; processing speed, e.g. system-user response time. The names of the metrics can be customised for an application area; for instance, the number of transactions per second of a Web server is often named *number of hits per second*, while for a telecommunication infrastructure it is named *call attempts per second*.

The metrics list, the complexity of the selected scenarios and the requirements for the connection with the SUT determine the requirements for selecting a test tool. However, the selection of an adequate tool depends on various factors such as costs, ease of use, report generation capabilities.

In the next step, the workloads are documented in a *performance test plan*. The test plan contains technical documentation related to the execution of the performance test: which hardware is used to run the test, software versions, the test tools and the test schedule. The test cases and the test data are then implemented and executed in a selected performance test tool. Part of the test plan is also the *performance test procedure* which is specific to the selected type of performance test such as volume, load, stress, benchmark, etc. For example, the performance test procedure of a volume test increases the number of users until the system runs out of resources, while the test procedure of a load test increases the number of requests per second until the system cannot hold the load anymore.

After test execution, the final step is to prepare a *performance test report* in order to analyse system performance and document the results. The resulting report includes performance indicators such as maximal supported load or supported number of users. Additionally, the report contains various graphs which display the performance metrics along the duration of test execution.

2.1.2 Workload Characterisation

The performance of a system is tested by monitoring the system while it is being exposed to a particular workload [Smi07]. Workload is sometimes also called *operational profile* [Alz04] for the simple reason that workload can be seen as a profiling activity of the system operation. The concept of workload was originally developed to define performance tests for processors and time-sharing systems but, nowadays, the concept is broader in means, being used in many areas such as Web applications, networking or middleware. The performance of a system cannot be determined without knowing the workload, that is, the set of requests presented to the SUT. The workload definition belongs to the performance testing process (see Section 2.1.1) and is a description of the test actions against a tested system.

It is important that the workload reflects how users typically utilise that system. Overloading an SUT with a huge number of requests tells us how robust the system is, but this kind of test does not reflect the performance requirements for normal usage of the system and it gives no information about the behaviour of the system in daily scenarios. Therefore, the workload should describe real-world scenarios taking into account social, statistical, and probabilistic criteria. Appropriate workloads depend on the type of system being considered and on the type of user and application. For this reason, researchers proposed different workload approaches targeting particular types of systems.

The activity to select and define a workload is called *workload characterisation* [AKLW02] and it has the goal to produce models that are capable of describing and reproducing the behaviour of a workload. There are different ways to characterise the workload. The simplest approach is based on collecting data for significant periods of time in the production environment [SJ01], [LGL+02]. These representative workloads are then used to determine system performance for what it is likely to be when run in production. One issue is that the empirical data is not complete or not available when creating new technologies [Bar04b]. In these situations, partial data can be collected from similar systems and may serve to create realistic workloads. Other approaches are based on modelling formalisms such as Markov chains [AL93], [BDZ03] or Petri networks [GHKR04] to deriving models for the workload which are then used to generate performance tests [AW94]. One technique, often used in practise, is to apply the load modelling concepts described in [SW01]. This method, called Software Performance Engineering, creates a performance model of the SUT and helps estimating the performance of the system. Unfortunately, the method is based on empirical data which might not be accurate enough for a good evaluation. The approach presented in [MAFM99] seems to be the most appropriate to the goal of this thesis. The paper proposes a methodology for characterising and generating e-commerce workload models based on a user state machine. The state machine is used to describe the behaviour of groups of customers who exhibit similar navigational patterns. For more details on workload characterisation techniques, a comprehensive reference is [EM02].

Workloads are classified into *real* and *synthetic* workloads [Jai91]. The *real workloads* are those workloads observed on a system being used for normal operations. In practise, these workloads are not controlled, they are just observed, while the performance is estimated on the base of collected logs. The main disadvantage is that they are not repeatable; thus they can characterise only particular evolutions of the system under certain load conditions but do not help determine the behaviour of a system under more load. *Synthetic workloads* are simulations of real conditions and can be applied repeatedly and be controlled by the tester. A synthetic workload has behind it a model of the real workload. It can be parameterised and be executed for different sets of data allowing the tester evaluate the performance of the tested system under different load conditions. Very often, the real workloads are studied in order to recognise certain patterns which are the used to create synthetic workloads.

In any performance test the workload should contain:

- *the workload data* - it is the data used by the test system to create users and instantiate users behaviours. The workload data usually contains also parameters to tune the load, e.g., number of users, requests per second, types of interactions.
- *the test behaviour* - a performance test runs parallel interaction activities with the SUT. Often, the interactions are transactions based, and, several transactions belong to a session, e.g., voice call. The test behaviour include precise descriptions of how these transactions and sessions are to be executed.
- *the design objective* - the design objectives define the QoS constraints to be validated during the test. Many actions fulfilled by the SUT are associated with time constraints which describe how long those actions should take. These constraints should be validated by the performance test and in cases where they are not satisfied, the translation or the session is considered as failed.

In [Jai91] some of the aspects to take into consideration when designing a workload are presented. These aspects describe how the system is used in real life and constitute the background for the performance test design methodology presented later in this thesis:

- *arrival rate* - the arrival rate describes the intensity of events received by the system.
- *resource demands* - the demands of resources describe the needs of the system in terms of hardware resources such as CPU, memory, disk, etc.
- *resource usage profile* - the resource usage profile defines the sequence and the amounts in which different resources are used in a system.
- *timeliness* - timeliness define the workloads changes in usage pattern in a timely fashion.
- *description of service requests* - the description of a service request consists of service request characteristics such as *type* of requests, *duration* of requests, etc.
- *loading level* - a workload may exercise a system to its full capacity, beyond its capacity or at the load level observed in real world.
- *impact of external components* - the impact of components outside the system may have a significant impact on the system performance. These kinds of external influences also should be simulated by the test workload.
- *repeatability* - the workload should be such designed that the results can be reproduced.

Besides these aspects, another important aspect, being essential for nowadays workloads, is to ensure that the workload is stateful session. Many protocol messages carry stateful information which has to be correlated with the information retrieved from other messages. In a *stateful* testing approach, the test behaviour defines a sequence of requests and settings for controlling the state maintained between them. Most performance tests define cookie-based or similar session tracking. *Stateless* workloads do not require to keep track of previous messages and, therefore, are easier to simulate. The tools which produce stateless workload are usually called traffic generators and have the purpose to only exercise the SUT with requests following a given pattern.

2.1.3 Performance Test Procedures for Different Performance Test Types

A performance test procedure defines the steps to be performed by a performance test. There are several variations of performance testing; each one regards specific facets of the performance of the tested system. The literature distinguishes: *load, robustness, stress, scalability*, or *volume testing*. For them, the performance test procedures are different.

Many sources elaborate on these definitions: “Testing computer software” by Kaner et al [KFN99], “Testing Object-Oriented Systems" by Binder [Bin99], “Testing applications on the Web” by Nguyen et al [NJHJ03], and “Software testing techniques” by Loveland et al [LMPS04]. However, people use these terms differently. The following definitions are used within this thesis:

Load testing [Jai91] simulates various loads and activities that a system is expected to encounter during production time. The typical outcome of a load test is the level of the load the system can handle but also measurements like fail rate, delays under load etc. Load testing helps detecting problems of the SUT such as abnormal delays, availability or scalability issues, or fails, when the

number of emulated users is increased. A load test defines real-life-like volumes of transactions to test system stability, limits or thresholds. Typically, a number of emulated users interacting with the SUT have to be created and managed while the functional behaviour of their communication with the SUT has to be observed and validated.

Scalability testing [Bin99] is a special kind of load testing, where the system is put under increasing load to determine how the system performance varies when the resources vary. For example, comparing an application while running on a machine with one processor versus running it on a machine with two processors. This helps to conclude how much the system performance increases in terms of requests per second.

Robustness testing [Bar02] is load testing over extended periods of time to validate an applications stability and reliability. During short runs of load tests or volume tests the system may behave correctly, but extending the testing period of time to a few hours, may reveal problems of the inspected system.

Stress testing [Bar02] is the simulation of activities that expected to be more “stressful" when an application is delivered to real users. Stress tests measure various performance parameters of the application under stressful conditions. Examples of stress tests are *spike testing*, e.g., short burst; *extreme load testing*, e.g., load test with huge number of users; *hammer testing*, e.g., continuous sending of requests. The main purpose of stressing the system is to make sure that the system is able to recover when it fails.

Volume testing [Bar02] is the kind of performance test, executed in order to find which volume of load an application under test can handle. For example, a client-server application is tested for the maximal number of users it can handle. It is, however, different from load testing since the volume testing does not necessarily imply that the load is very high.

Benchmarking is a performance test used to compare the performance of a system [Men02a]. Benchmarking is considered in general to be a load test which has a well-defined execution procedure that ensures reproducible results.

2.1.4 Performance Measurements and Performance Metrics

During the execution of a performance test, *performance measurements* are performed. Measurements are collections of events with timestamps [SSR97]. The events mark different events such as creation and termination of a transaction. The measurements are then used to compute performance metrics such as resource consumption, latency or throughput. Performance metrics are used to derive more elaborated performance characteristics such as mean, standard deviation, maximum and minimum or their distribution functions.

Performance characteristics can be evaluated either off-line, after the performance test finished its execution, or on-line, during the performance test execution. The on-line approach requires to evaluate the performance constraints on the fly. The constraints are sometimes also called performance design objectives. They define requirements on the observed performance characteristics, e.g., the response of a request should arrive within 50 ms.

2.1.5 Performance Test Architectures

A widely used approach for testing is the Conformance Testing Methodology and Framework (CTMF) standardized as International Standards Organization (ISO)/International Electrotechni-

cal Commission (IEC) 9646 [ISO94]. The CTMF defines several test architectures, called abstract test methods, and the Tree and Tabular Combined Notation (TTCN) (ISO International Standard 9646) [BG94]. Unfortunately, the CTMF proved to be not good enough to cope with performance testing needs. Therefore, an extension of CTMF with performance testing concepts has been proposed [SSR97]. The extension proposes an approach for testing the performance of communication network components such as protocols, services, and applications. In that approach the test system is seen as a distributed system that consists of active components, called test components. The paper also proposes a performance test configuration for an end-to-end service which is depicted in Figure 2.1.

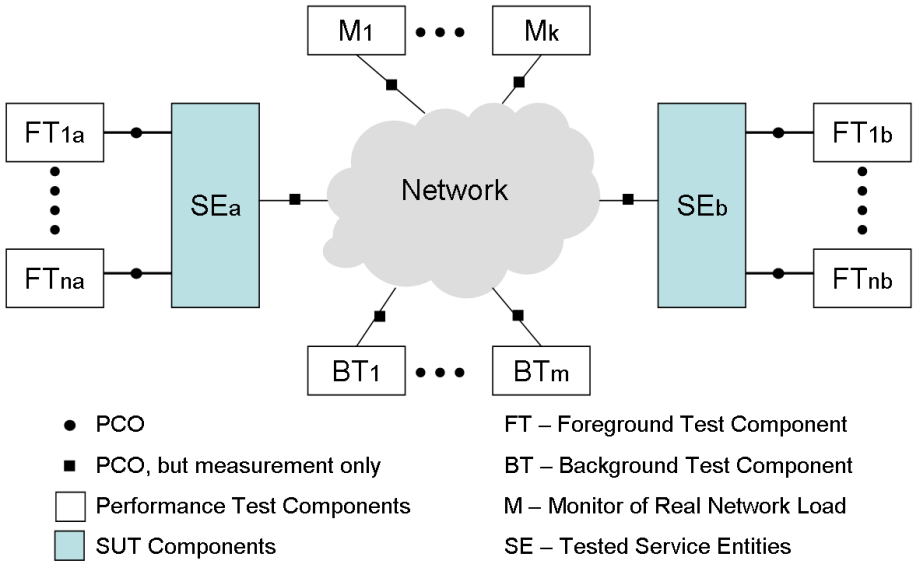

Figure 2.1: Interaction between the Performance Test System and the SUT

The test components are of two types: *foreground* test components, which communicate with the SUT directly by sending and receiving Protocol Data Units (PDUs) or Abstract Service Primitives (ASP)s similar to conformance testing, and *background* test components, which use load generators to generate continuous streams of data but do not interact with the SUT. The foreground component are used in fact to emulate the clients. A multi-service SUT would consist of many Service Entity (SE)s. Points of Control and Observation (PCOs) are access points for the foreground and background test components to the interfaces of the SUT. They offer means to exchange PDUs or ASPs with the SUT and to monitor the occurrence of test events. Coordination Points (CPs) are used to exchange information between the test components and to coordinate their behaviour. The test components are controlled by the main tester via coordination points.

This architecture is referred in [WSG98] as part of a survey of various test architectures for testing including performance testing. The paper proposes also a general test architecture, which can be adapted to different types of testing and to testing of applications based on new architectures and frameworks.

The test architecture concepts have been further extended to the TTCN-3 Runtime Interfaces (TRI) interfaces in [SVG03] and TTCN-3 Control Interfaces (TCI) interfaces in [SDA05]. The TRI provides a set of operations to adapt the abstract methods to the execution platform and to the network.

These concepts gained a more general interest from industry and have been adopted as an ETSI standard [ETS07b]. The TCI interfaces consist of operations to handle the test data and the communication between test components. They also have been standardised by ETSI in [ETS07c]

2.2 Performance Test Framework

This section discusses design requirements for the functional architecture of the performance test framework. Many commercial or non-commercial test frameworks are available. In [WSP+02] the possibility to integrate them by means of a unified set of data exchange, control and Graphical User Interface (GUI) interfaces is discussed. However, not all of them may offer a suitable framework to develop performance tests based on the concepts introduced before.

2.2.1 The Functional Architecture of a Test Framework

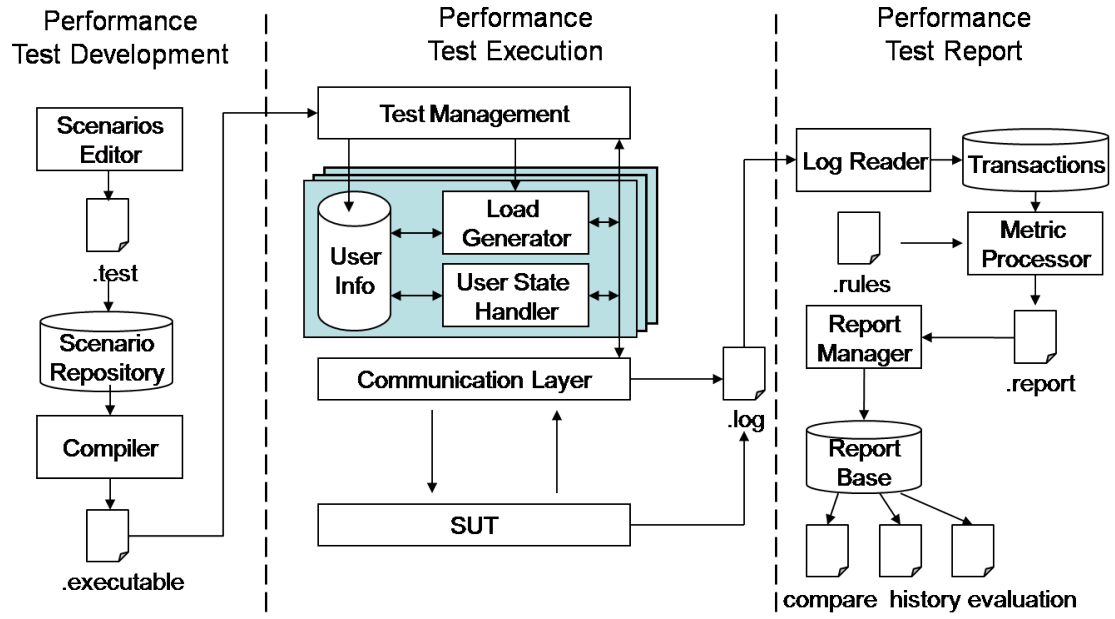

Figure 2.2: Architecture Functional Blocks

The functional elements are depicted in Figure 2.2. They are structured into three main blocks: *test development, test execution* and *test report.* The architecture takes into account the realisation of test systems capable to satisfy testing needs in terms of developing, executing or analysing performance. However, this set of requirements represent a minimal specification which can be extended with further features.

The development and execution are closely linked. The test development not only implies the editor and test files management, it also includes the libraries, data representation and connectivity to SUT. In general, the more functionality is offered in form of libraries, the easier the test development is. Ideally from the test development point of view, the tester should not care about the test

distribution, the connectivity to SUT, the call arrival pattern or the data encoding. In this respect, the development should only concern the realisation of the test behaviour and testdata [ER96].

Test artefacts have also to be *maintained*. This is usually supported in test frameworks by a *test repository*. The test behaviour may be interpretable code or binary code (in this case a compiler is needed). In either case, the test behaviour is deployed onto an execution framework. The execution includes *test management* functionality which takes care of scheduling of runs, test parametrisation and monitoring. Another major requirement for execution is the *throughput performance* which determines how many messages per second the test system can handle. The throughput is usually improved by scaling the performance through load distribution, by moving to better hardware or by finding better algorithms for critical parts of execution which are also called hot-spots.

The test execution is realised by a *load generator*, several *user handlers* and a *user repository*. The load generator provides ready-to-use traffic-time profiles which have to be parameterised with the workload. Part of the workload is the user data which is maintained within the user repository. There is an entry for each user which consists of the identity and the state. The load generator selects users from the user repository and instantiates test behaviours. The user handlers are responsible for handling the events received from SUT.

During the execution, the interaction between users and SUT is monitored in the form of events with timestamps. Also the resource consumption on SUT side is monitored. These measurements are the input of the *performance test report* part. A *metrics processor* correlates the measurements into metrics and gives them a human readable representation in the form of graphs and charts. For more flexibility, the metrics processor should provide an easy to use front-end which allows the specification of the metrics. A report manager should also be available in order to manage the results obtained for different test configurations, hardware configurations, software versions, etc.

2.2.1.1 Performance Test Development

The tests have to be written into a machine-understandable language in order to make them executable. Test development is used within this thesis to refer to the activity of programming tests. They can be either manually written or be generated in a programming language. Sometimes, test specification term is also used in this thesis. This term refers to the abstract view of a test which contains the specification of the data input, interactions and expected results. The test specification does not necessarily have to be executable.

Flexible data representation. The communication protocols used today for the communication between UEs and SUT involve many types of messages with rather complex structures. Additionally, the message fields may be defined optionally or may depend on the values of other fields. To cope with these requirements, the test development framework must assure that any message structure or constraint can be described in an easy way.

Event handling. At test execution many messages are interchanged between the test system and the SUT. The test system has to handle each event occurred as effect of receiving of a message from the SUT. Typical operations are: queuing of messages, finding the user to which an enqueued message corresponds, updating the state of the user, acting appropriately by sending a new message or closing the transaction. Such operations have to be easy to define within the test framework.

User state handling. In general, any message received or sent by a user changes the state of that user. For any message, the test system has to update the state of a user. Therefore, a user state description as well as a state update mechanism has to be available in the test framework too. Furthermore, the mechanism has to be both performant and easy to use.

Test parametrisation. The test system should provide the possibility to define test parameters. Additionally, the test parameters should be accessible within test implementations.

2.2.1.2 Performance Test Execution

Test execution takes place after the tests have been developed and are ready to run in the target hardware environment. At the *execution of performance tests*, similar to the execution of functional tests, the test system applies test stimuli to an SUT while the behaviour of the SUT is monitored, and expected and actual behaviours are compared in order to yield a verdict. However, different from functional testing, performance testing runs parallel tests [SDA05]. Technically, each test is a separate process and it simulates the behaviour of one SUT user [2].

Code deployment. The first step of test execution is the *code deployment*. It sums up the operations to prepare the test behaviour for execution. For example, in a Java [SM08] based environment, a test behaviour in the form of a Java archive is loaded in the test tool via reflection mechanism.

Load generation. The test system must be able to execute test scenarios following a traffic-time profiles configuration. The shape of traffic-time profile must be supported as a reusable mechanism by the test framework itself and its configuration should be possible through test parameters. The precision of the load generator must be good enough to satisfy the temporal constraints imposed by the event arrival distribution used for traffic generation.

Test behaviour execution. Test behaviour execution is the concrete execution of the interactions between the test system and SUT. Every action or stimuli from the test system is encoded into a protocol unit and is sent over the network link to the SUT. Corresponding responses, received back from SUT are validated and further actions are executed.

Test distribution. In order to increase the performance of a test system, parts of the test behaviour can be distributed over several test nodes. At the physical level, parallelism is realised by using multiple CPUs and/or multiple interconnected nodes. Therefore, the assumption that the workload execution involves several distributed computing devices is made. Additionally, each test node may contain more than one CPU. Distributed testing is the testing activity which involves a distributed test environment. It combines two basic concepts - workload distribution and remote control - in order to offer highly scalable and flexible test systems. In a distributed test environment, several test systems running on different nodes are controlled from one or more remote control centres. The multiple test systems can be combined logically to operate as a single unit.

Load balancing. The test distribution is usually correlated with load balancing techniques applied to efficiently use the underlying hardware environment. Load balancing algorithms are used to minimise the execution time, minimise communication delays and/or maximise resource utilisation.

Network emulation. The test system should also emulate the network characteristics of the different interfaces as long as users may use different interfaces to connect to SUT. This includes

[2] Parallelism is realised by simulating the parallel behaviour of a large number of SUT users.

network bandwidth, latency and error rate. These characteristics are to be set separately for each direction so that non-symmetric interfaces can be emulated, e.g., up and down bandwidth on a DSL link. These settings should be supported by a *communication layer* which provides a transparent interface with respect to user behaviour.

Monitoring. Monitoring implies generation of large amounts of measurements which are later used to analyse the performance of the SUT and generate performance test reports.

Synchronisation. Test distribution is a major requirement for scaling the performance of a test system. In the case that the test framework supports distribution over several test nodes, a synchronisation mechanism is needed. The mechanism is used to pass control information between the test nodes.

2.2.1.3 Performance Test Report

The performance test report documents the test results. It consists of monitored data represented as charts and data sets which help to understand the behaviour of the SUT over the elapsed time of the test. Additionally, the test report presents the relevant metrics that are conventionally used to compare test results.

Test reports management. A detailed performance evaluation requires the execution of multiple test runs with different test configurations, test hardware or SUT software versions. This will result in a large number of reports. To control a large number of reports, the test report tool should offer a management feature to track/retrieve the reports on different criteria: time, SUT version, SUT hardware, etc.

Illustration of history logs. The history view is also a test result management requirement which allows to investigate the performance increase/decrease of an SUT along different versions of the software. This view may vary from a simple one, which reports only the variation of some metrics, to a more complex one, which displays charts with the variation of communication latency, round-trip times, resource consumption, etc.

Comparison view. The comparison view is similar to the history view, but the comparison is realised for only two sets of results with the same test configuration, e.g., same software but different hardware. The comparison view should display as many differences as possible including resource consumption graphs, latency, error rates, etc.

2.2.2 Towards Test System Performance

The intensive use of hardware resources, e.g., 100 percent CPU usage, during the test execution leads very often to malfunctions of the test system which ends up running slower than expected. Consequently, the test results can be wrong as an effect of erroneous evaluation of SUT responses. Such a situation is encountered when, for example, the test system creates too many parallel processes which share the same CPU. The processes wait in a queue until they acquire the CPU according to the used scheduling algorithm. Hence, the bigger the number of processes is, the more time a process has to wait in the queue until it acquires the CPU. Since the execution of critical operations such as timer evaluation, data encoding or decoding is automatically also delayed, the test system may consider an operation timed out while, in reality, it did not. The same phenomenon has a considerable impact also on the load generation by decreasing the number of interaction per second.

2.2.2.1 Evaluation of Test System Performance

The evaluation of performance test systems [GMT04] turns into a problem of determining whether the SUT is that slow as the results reveal or rather the test system is overloaded by its testing activities and cannot produce the necessary load and/or reacting in time. Answer to this question can only be given after analysing the quality of the test execution. To detect such problems, several parameters which help the tester to validate the test execution are observed.

One of these parameters is the duration of the execution of critical test tasks. It assigns temporal dimensions to all operations to be executed sequentially in a test which might influence the evaluation of SUT's performance. For example when receiving a message from SUT and this message is used to validate the reaction of SUT to a previous request, the test system has to decode and match the received message only in a small amount of time, otherwise the additional computation time will be counted as the SUT reaction time.

A further interesting parameter is the quantity of the demanded resources. If the test system constantly requires the maximum of the resources the underlying hardware can allocate, this is a first sign that the test might not be valid. Another parameter is the deviation average from load shape. If the load does fluctuate very often moving from lower to higher values, it proves that the test system might be overloaded.

A performance test is considered to be valid only if the platform satisfies the performance parameters of the workload. The quality of the load test execution is guaranteed if the test tool fulfils the performance requirements with respect to throughput and latency.

2.2.2.2 The Performance Pyramid

Looking at the test system from a performance perspective, requires a good understanding of the layers of a test system architecture. Figure 2.3 is an approach to thinking about the performance of a test system. This is an adaptation to test execution of the model presented in [HM01].

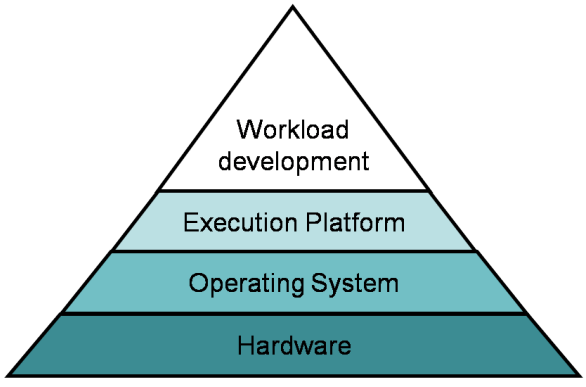

Figure 2.3: The Performance Pyramid

In this view, the test system is represented as a pyramid with several layers. Each layer plays an extremely important role regarding the performance. If one understands how things work at each level, one can better decide upon the load distribution strategy or decide how to tune the performance of the overall system. However, one important aspect of this performance pyramid is that

as the performance problems have been improved at one level, other performance improvements can be still realised at other levels.

The physical architecture of a typical distributed test execution environment is an assembly of computing devices. The *computing device* concept covers anything one might use to run test activities such as general purpose PCs, specialised testing hardware with SUT specific connection hardware, or with protocol specific hardware stack or real-time computation systems. Any of these physical computing devices is called a *test node*. The test nodes are connected by physical *communication channels*. The connections may be of various types and technologies and typically include network connection devices such as routers or switches and Unshielded Twisted Pair (UTP) cables.

Additionally, the test environment may include more specialised hardware test devices, which do not execute test behaviours. They actually provide an operation interface to the rest of the environment for external services, e.g., Domain Name System (DNS), time server.

Hardware. This is usually the easiest part to be improved in terms of performance, since in general, putting better hardware in the environment lets the system perform better. However, upgrading the hardware is not enough without understanding how it works in relation with the applications running on it, in order to make the best of the current environment. This covers a range of elements, including:

- *communication factors* - such as the speed and physical components, and the access speed of disk storage.
- *the overhead of switching environments and data transforms* - required between environments.
- *advantages of using multi-core processors* - that are the latest innovation in the PC industry. These multi-core processors contain two cores, or computing engines, located in one physical processor and present the advantage that two computing tasks can be executed simultaneously.

Operating System. The OS processes execution platform requests for hardware resource allocation and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. The OS is typically the place where many test systems can be easily tuned for better performance. Additionally, the OSs offer also configurable load balancing strategies for the parallel processes managed by that OS. In general-purpose operating systems, the goal of the scheduler is to balance processor loads and prevent one process from either monopolising the processor or being starved of resources [LKA04].

Execution Platform. The execution platform provides the execution engine for the workload specification. There are many tools for performance test execution and, consequently, many strategies to execute a workload. The programming language used to develop the execution platform may also influence the way the execution platform is designed. The most popular languages used to implement execution platforms are C/C++, Java, C#.

Workload Development. Most tools provide programming or scripting languages to ease the workload development. These languages offer abstract means for parallelism, e.g, parallel components, threads, processes; for data encapsulation, e.g., messages, hashes, tables or even database access; and for network connectivity, e.g., channels, ports.

For each language, one has to understand how these abstract elements are *transformed* into elements of the execution platform layer. For example, a parallel component from the programming language can be transformed into a single execution platform thread or can be grouped with other parallel components into a bulk of components attached to a thread. The performance of a system decreases when a few thousands of processes are running in parallel. In the first mapping strategy the system will suffer of performance earlier than the second mapping strategy. Therefore, in the first situation, it is good to choose a better way, though more complex, to program the workload.

The tools which do not offer such an abstract specification layer provide a platform level programming Application Programming Interface (API). The test designer then has the task to *program* rather than to *specify* the test. Such an example is JUnit [JUn07] testing framework and its related extensions.

2.3 Summary

This section introduced concepts on performance testing. Depending on the applied test procedure several types of performance tests are identified: load, stress or volume tests. The design of a performance test requires several main parts: workload, metrics, load parameters. A performance test is then implemented and executed in an execution platform. The execution platform supports logging of the execution and based on this log traces the performance of the tested system can be analysed. Sometimes the performance of a single test node may not satisfy the load requirements, then distributed test environments are employed. Test distribution brings along load balancing techniques and methods to efficiently design performance tests for the running of distributed execution frameworks.

Chapter 3

Performance Testing Methodology and Realisation Patterns

All truths are easy to understand once they are discovered;
the point is to discover them.
– Galileo Galilei

Although performance testing is a common topic among people and organisations, the research does not address performance testing of multi-service systems at a general level but rather targets only specific types of applications, e.g., Web applications [MAFM99]. This chapter introduces a methodology for developing performance tests for multi-service systems including the methods to design the performance tests and a selection of implementation patterns.

3.1 Multi-Service Systems

Before discussing the performance testing of multi-service systems, the definition of the multi-service system must be given. A typical multi-service system offers a number of services which can be accessed through entry-points [MP02]. In this context, a service runs on a service platform [Cas04], [MP02] which allows organisations and individuals to serve and consume content through the Internet. There are various levels of services and various kinds of services offered.

The system consists of many sub-systems (hardware and software) communicating usually through more than one protocol. The sub-systems provide different functionalities and either host services or mediate their access and administration. The service consumers are the end users which access the services through compatible devices called UEs. The service consumption is realised through communication protocols involving different types of transactions, e.g., authentication, charging. With the specification of IMS, the current telecommunication infrastructure is moving rapidly toward a generic approach [CGM05] to create, deploy and manage services; therefore, a large variety of services is expected to be available soon in many tested systems.

Beside this service variety aspect, two more aspects are visible too: the service complexity and the randomness in services demand. The services are becoming more and more complex, requiring more computing resources on the system side and more messages exchange between involved components. The expectation is that the more complex and demanded the service is, the more the system performance decreases. The service demand randomness concerns the user preferences

for services and describes the way users may select a service from a list of available services. As result, the overall system load is a composition of instances from different services where each type of service contributes in a small proportion.

The services are deployed within a network of components, where each component has a well-defined functionality, e.g., authentication, registration, charging. To gain more performance, some of the components may be even distributed over several servers. When a service is invoked by a user, these components interact with each other through messages exchange. The communication requires a good synchronisation between the components and a reliable service state handling mechanism capable to work under heavy loads. Among the well-known issues met when evaluating the performance of a complex distributed system, one has to take into account that not all software components are proprietary. Due to the complexity of the present systems it is expected that some of the components are third-party components. Therefore, during performance testing it may be discovered that these components do not interact properly with the rest of the system, or that some of them are performance bottlenecks for the whole system. However, the challenge remains the same: to find the performance of the whole system and, eventually, be able determine which component is the bottleneck for the performance.

The services may be available on two or more networks. These networks may interconnect with each other such that users from one network may communicate with users from other networks. This aspect has to be considered too while designing performance tests. The TS should be able to demand in adequate proportions also services from foreign networks. However, in this configuration, the TS is required to simulate users which connect to all networks.

Another characteristic is that different users exhibit different navigational paths for the same service. Typically, users follow the interaction path for a successful call but also failure paths may happen, e.g., user not available, abandoned calls. Depending on the service, the communication between two entities varies in complexity. The more complex the service interaction is, the more failure scenarios may occur.

3.2 Performance Testing Reusing Functional Tests

One of the well-established methods in testing is that of functional testing [SPVG01]. The functional testing of multi-services systems is used to check if the services meet functional requirements, i.e., that the service is functionally correct.

Along this thesis, performance testing is seen as an extension to functional testing to check performance requirements. The basic idea is to define performance tests by reusing basic functional tests for the services [SDA05]. Test components are used to emulate service clients. These test components perform basic functional tests to evaluate the reaction of the services to their requests. The combination of test components performing different basic functional tests in parallel leads to different test scenarios for the services. Parameterisation of this test framework enables flexible test setups with varying functional and performance load.

3.3 The Performance Test Design Process

For testing multi-service systems, there is an emerging need for a more general methodology which focuses on the user-service relation. The methodology should address both sides of the

problem: functionality aspects, e.g., user behaviour and non-functional, e.g., concurrency, QoS. To address these aspects a number of new concepts have been introduced. The performance test design methodology is based on the performance test design process which is depicted in Figure 3.1. This process refines the general process described in Section 2.1.1 as applied to multi-service systems.

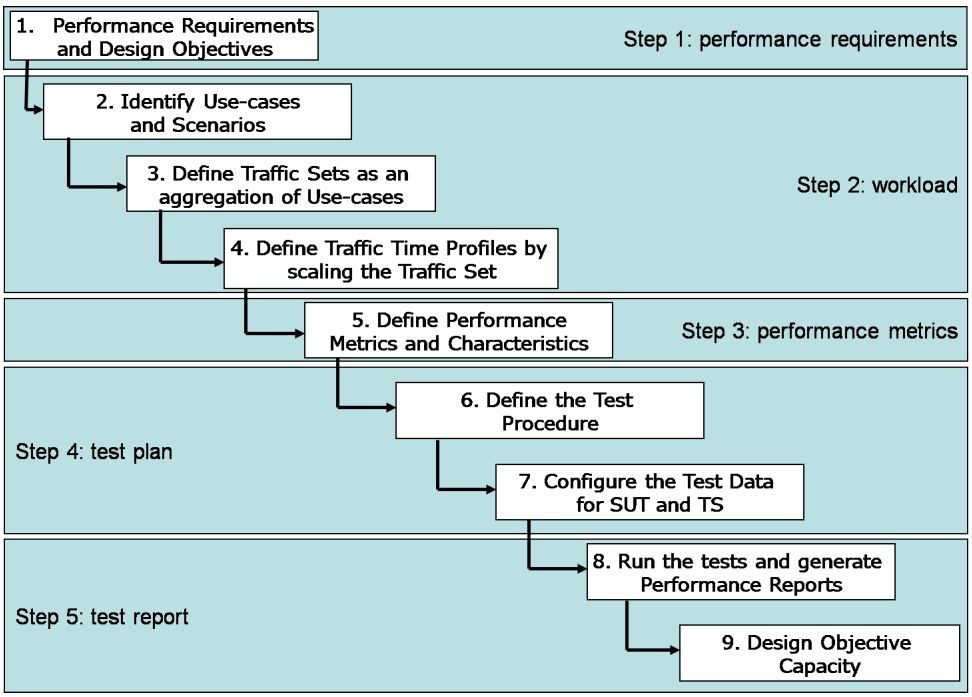

Figure 3.1: The Performance Test Design Process

Performance Requirements. The process starts with the selection of *performance requirements*. In this step the test engineer has to identify which features characterise the performance of the tested system. High number of transactions per unit of time, fast response times, low error rate under load conditions are examples of common performance requirements. But there are also other specific requirements which might be considered for a system in particular: availability, scalability or utilisation. A detailed view on performance requirements selection is provided in [Nix00] where a performance requirements framework integrates and catalogues performance knowledge and the development process.

Workload. The second step is to define the *workload*. The workload consists of the *use-cases* which are the types of interactions between users and SUT to access services, e.g., voice call, conference call, application call. Since a multi-service system provides many services for each use-case, it is required that the workload is created as a composition of multiple test scenarios from each call model. This task requires a systematic search through the SUT's specification documents in order to recognise all possible interaction models. A concept which extends the previous performance test process is the traffic set which enables the possibility to parameterise the traffic composition instead of having a fixed one. This way, multiple traffic sets can be defined, e.g., at city, region, country level, taking into account social and technological factors. The workload is also associated with a predefined traffic-time profile which combines the traffic sets with existent traffic patterns, e.g., random call arrival rate.

Performance Metrics. In third step, the performance metrics have to be defined. Different to related works, for performance testing of multi-service systems it is necessary to gather information about each scenario. Therefore, two types of metrics are regarded: global metrics similar, e.g., CPU, memory, fail rate, throughput, and scenario related metrics defined for each scenario, e.g., fail rate, latency. The performance metrics list should include the key performance indicators which are relevant for the overall evaluation.

Test Plan. In forth step, the SUT and the TS are parameterised by the primary provisioning parameters, e.g., user identities. The performance tests are executed according to a performance test procedure which takes into consideration the effects of long duration runs involving many services.

Test Report. The final step is to execute the tests according to the test plan and generate reports with meaningful data, e.g., largest sustainable load, number of supported users. The related works provide few details about the structure of a performance test report. Therefore, this thesis goes into more detail with respect to the structure and contents of a test report.

3.4 The Performance Test Concepts

The performance test elements are presented in Figure 3.2. The concepts are grouped into three main blocks:

- *use-cases* - use-cases describe the types of behaviours of an individual user, and which in turn define scenarios.
- *performance tests* - scenarios from different use-cases are used in a performance test which generates a workload by aggregating the behaviour of individual scenarios in a controlled manner and collects log-files of measurements during the test.
- *performance test reports* - at the end of the test, the test report is created in order to report the metrics correlated from the log files.

A scenario is defined as a message flow between the interacting parties. For each scenario, *metrics* and *design objectives (DOs)* are defined.

A *performance test* combines scenarios from different use-cases into a *traffic set*. Within a traffic set, each scenario type is instantiated in a predefined proportion, interpreted as its probability of occurrence during the execution of the performance test procedure. These proportions are specified in a traffic-time profile.

The *performance test report* is generated after the execution and it contains a full description of the SUT configuration, the TS configuration, the process used to generate the system loads at each SUT reference point, and data series reporting the metrics as a function of time.

3.4.1 Use-Cases and Scenarios

The first step in the workload creation is to identify the *use-cases* and, for each use-case, select multiple *test scenarios*[1]. A use-case is usually associated to one service, however, a use-case may be defined also as a composition of services. *Use-cases* define interaction models between one or

[1] In the rest of the thesis, the term test scenario is mixed with the term *scenario* and refer the same concept.

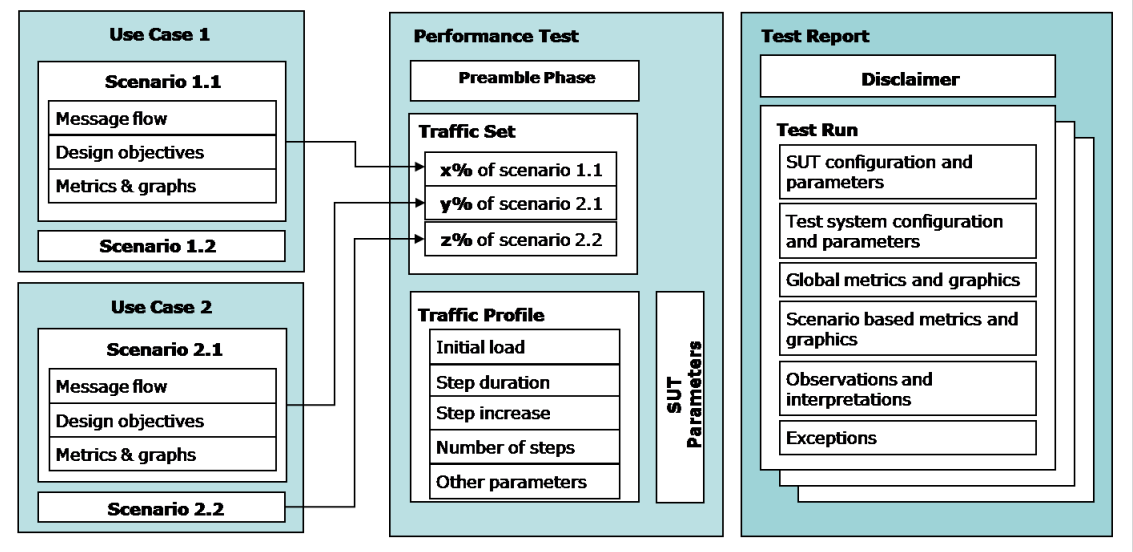

Figure 3.2: Performance Test Concepts

more users and the SUT, for instance, in the telecommunication domain, there are services such as voice call, conference call. Different to test scenario, a use-case is a specification of a general type of interaction between a TS and a SUT, corresponding to a mode of end-user behaviour.

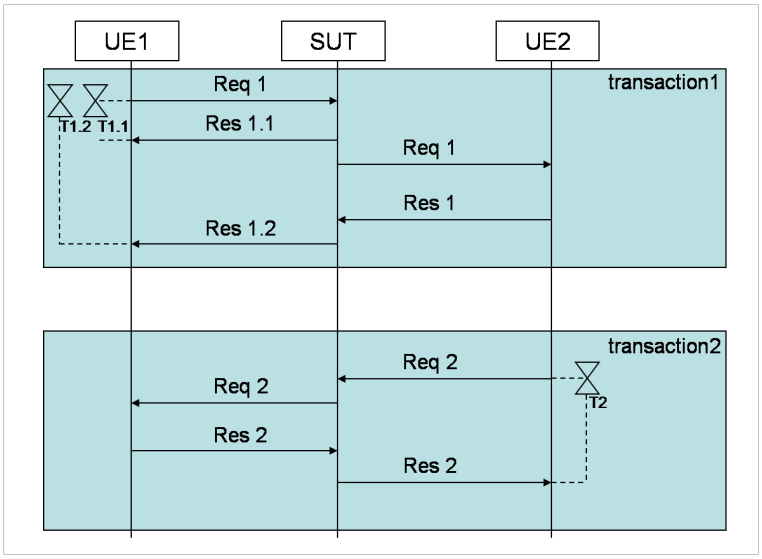

Figure 3.3: Test Scenario Flow Example

An individual interaction path is called a *test scenario* and describes a possible interaction determined by the behaviour of the user and other system actors. The typical questions that have to be answered at this point are: Which services are going to be used most? Which particular flows are characteristic to a given scenario? What is most likely to be an issue? Each test scenario is described by its message flow between the talking entities.

An example of a test scenario is depicted in Figure 3.3. This example presents the interaction between two UEs and the SUT. All entities of the SUT are represented as a single box. The interaction is based on request/response transactions. For each request, there is at least one response. The flow presented in the figure consists of two transactions. The first transaction is initiated by the UE1 which sends the Req1 request message to the SUT. There are two responses for this transaction. The first response is sent by the SUT while the second is sent by the UE2. The second transaction is created by UE2 after an arbitrary period of time consumed after the first transaction. In that transaction, the UE2 sends the Req2 which is answered by the UE1 with the Res2 message. Each response has to be received within a time limit. This time is modelled as a timer which measures the time spent between the request and response. If the response is not received within the expected time, the transaction runs into a fail situation.

As presented in the previous chapter, it is considered that the more the workload approximates the reality, the more meaningful the performance test results are. To most representative workload is realised when the selection of test scenarios covers all possible interaction flows including positive and negative flows. The selected test scenarios should capture typical situations such as success or fail as encountered in reality. In order to achieve a good coverage of test scenarios one has to regard the following types of scenarios. They are exemplified with the help of the simple voice call service as it is presented in the telecommunication domain:

- *successful scenarios* - this type of scenario happens most of the time and should always be included in the scenario list of each identified use case. A voice call is called successful when all transactions are completed without any error states caused, for instance, by timeouts, wrong content or IDs, etc.

- *fail scenarios* - many causes may lead to a fail of a service. For the voice call service example, a typical error situation occurs when a user tries to call another user who is not available. The TS simulates this scenario by creating a call to a user who does not exist.

- *abandoned scenarios* - this test scenario is based on an abandoned service interaction. The TS should simulate users who abandon the calls before they terminate. For example, a user initiates a voice call to a second user, but before the second user answers, the first one cancels the service interaction.

- *rejected scenarios* - in a service involving more than one user, it may happen that one of the users - not the one who initiates the service - declines the service request. In the voice call example where one user calls a second user, the rejected scenario happens when the second user declines the call.

In Figure 3.4 the state machine of UE1 is presented. A general pattern to describe the state machines of users which deal with telecommunication services can be described as in this example; the behaviour of any user involved in a test scenario can modelled in a similar way. The UE1 starts from a state called *available*. A user is *available* when it can be used to create a new call. There are three types of actions which may change the state of that user:

- *external actions* - these actions are triggered by the load generator which controls all users. The load generator may decide that a user starts a new call scenario and does this by triggering an action to start the scenario.

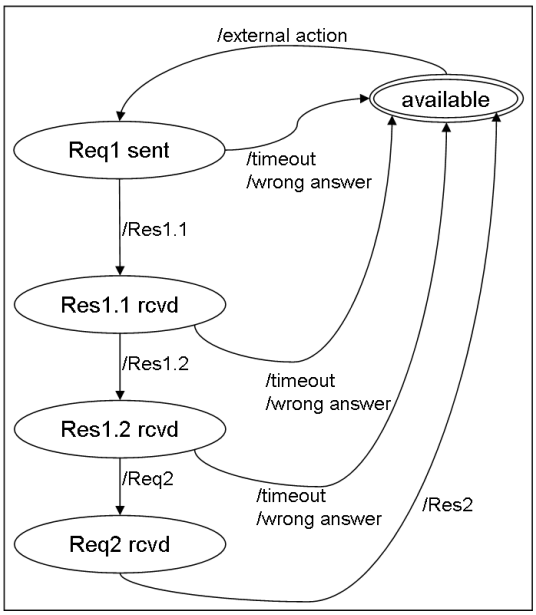

Figure 3.4: User State Machine Example

- *communication events* - these actions correspond to messages received from SUT or sent to SUT. The messages may be *correct* messages or *incorrect* messages. In either case the user should react. Therefore the state-machine should model also these kind of exceptions.
- *timeout events* - for each request, a transaction timer is started. It expires after a given period of time. A timeout event implies that no message has been received for the associated transaction.

The UE in Figure 3.4 starts a new call scenario at an external action request issued by the load generator. The UE creates the first transaction and sends the first request message. If the SUT answers with Res1.1 message, the UE goes into *Res1.1 rcvd* state. Otherwise, if a timeout occurs, or if the SUT replies with another message type, the scenario is considered as fail and the UE goes back into the *available* state, which means that the user may be used again for another call. From *Res1.1 rcvd* state, the user goes into *Res1.2 rcvd* state if the Res 1.2 message is received.

The two transactions are independent and are executed at different points in time. This means that the user may wait for an arbitrary amount of time until the second UE sends the Req2 message. If the UE2 or the SUT sends a different type of message instead of Req2, the UE1 goes into *available* state. However, in order to ensure that the UE1 will finish the state machine, a timer expiring after a sufficient amount of time may be started. If this timer expires, it means that the second transaction will never take place. If the Req2 is received, the UE1 goes into *Req2 rcvd* state. From that state, the UE1 goes into *available* state, after sending the Res2 response.

3.4.2 Design Objectives

The performance evaluation is based on two steps: firstly, collect measurements about each instantiated test scenario during the workload execution and secondly, derive the performance metrics

to evaluate performance requirements. The collected measurements consist of logging events of each message interchanged with the SUT. Additionally, the timestamp when the message becomes visible to the TS is logged together with the message.

The performance requirements concern error rates and delays. They are evaluated separately for each use-case on top of the collected measurements. For each performance requirement a Design Objective (DO) is defined as the threshold value which is then used to compare the metrics.

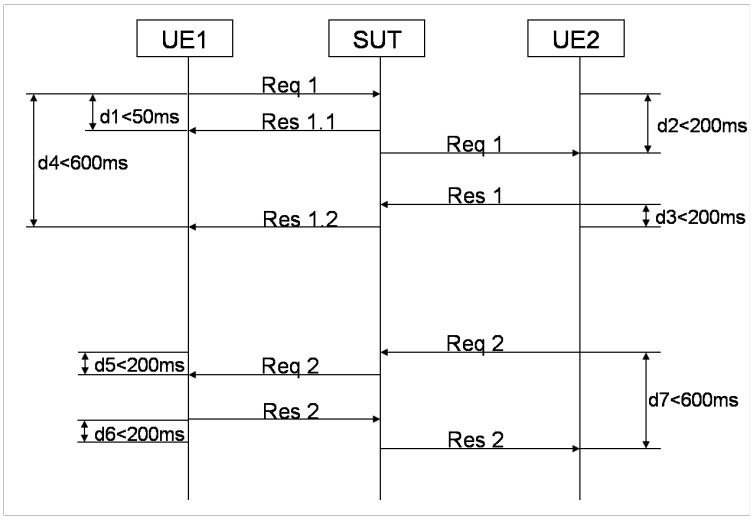

Figure 3.5: Design Objectives

DOs for Delays. Figure 3.5 depicts one scenario flow including the associated DOs for communication delays. The DOs for delays are of two types:

- *latency DO* - defines the maximal time required to get a message through the SUT network from a caller UE1 to a callee UE2.
- *transaction round-trip time DO* - defines the maximal time required to complete a transaction including all messages.

These DOs define temporal limits for the latency of the SUT and can be modelled in the user state machine (see Figure 3.4) as timer events. When a timer expires, it means that a DO for that test scenario has been exceeded and the test scenario will count as a failed test scenario.

DOs for Error Rates. A DO for the error rate sets the threshold for allowed percentage of errors out of the total number of scenarios. Different than DOs for delays, the DOs for error rates should be set at use-case level such the error rate is validated for all types of scenarios which belong to that use-case.

3.4.3 Traffic Set Composition

The TS applies a workload to the SUT which consists of the traffic generated by a large number of individual simulated UEs. Each UE performs an individual scenario. Obviously, different scenarios may be combined in the same workload. A conceptual question is how to allow test engineers define compositions of scenarios. The concept to cover this aspect is called a *traffic set*.

Table 3.1: Traffic Set Composition Example

Scenario Type	Scenario Label	Scenario ratio	Scenario distribution
registration	s1.1	5%	poisson
re-registration	s1.2	20%	poisson
de-registration	s1.3	5%	poisson
successful voice-call	s2.1	21%	poisson
abandoned voice-call	s2.2	4%	poisson
rejected voice-call	s2.3	4%	poisson
fail voice-call	s2.4	1%	constant
successful page-mode messaging	s3.1	38%	poisson
failed page-mode messaging	s3.2	2%	constant

A traffic set example is presented in Table 3.1. It consists of a mix of nine scenarios selected from different use cases. The names of the scenarios are not relevant at this moment; they are only used to illustrate the concept.

Within the traffic set, each scenario has an associated relative occurrence frequency which is interpreted as its probability of occurring during the execution of the performance test procedure. This frequency indicates how often a scenario should be instantiated during the complete execution of the performance test.

As long as in reality the load is random, to avoid constant load intensities, the scenarios are instantiated according to an arrival distribution, which describes the arrival rate of occurrences of scenarios from the traffic set. The arrival rate characterises the evolution of the average arrival rate as a function of time over the duration of the performance test procedure. An example of such an arrival process is the Poisson process [NIS06] employed often in simulations of telecommunication traffic.

A histogram example of scenario attempts distribution, as resulting of a performance test execution, is presented in Figure 3.6. The graph displays the frequencies of the various load intensities values as scenario attempts per second, which occur along the test execution. For example, the load intensity value of 86 on X-axis has an occurrence frequency close to 0.4 on Y-axis. The curve shows that the frequency of values is higher around the mean and lower for values more distant to the mean, which corresponds to a Poisson distribution.

Though the traffic set concept allows any combination of scenarios, a relevant traffic set is a collection of scenarios which are determined to be likely to occur in a real-world situation. A source of information for traffic set selection is offered by telecoms statistics collected over the years. However, these statistics are unfortunately limited in space and time. They comprehend only a few scenarios since, in the past, very few services have been offered. The statistics are also not representative anymore for the current social context since user population has increased, services are cheaper, etc. This thesis does not consider the aspect of traffic set selection but offers a concept to experiment with any possible combination.

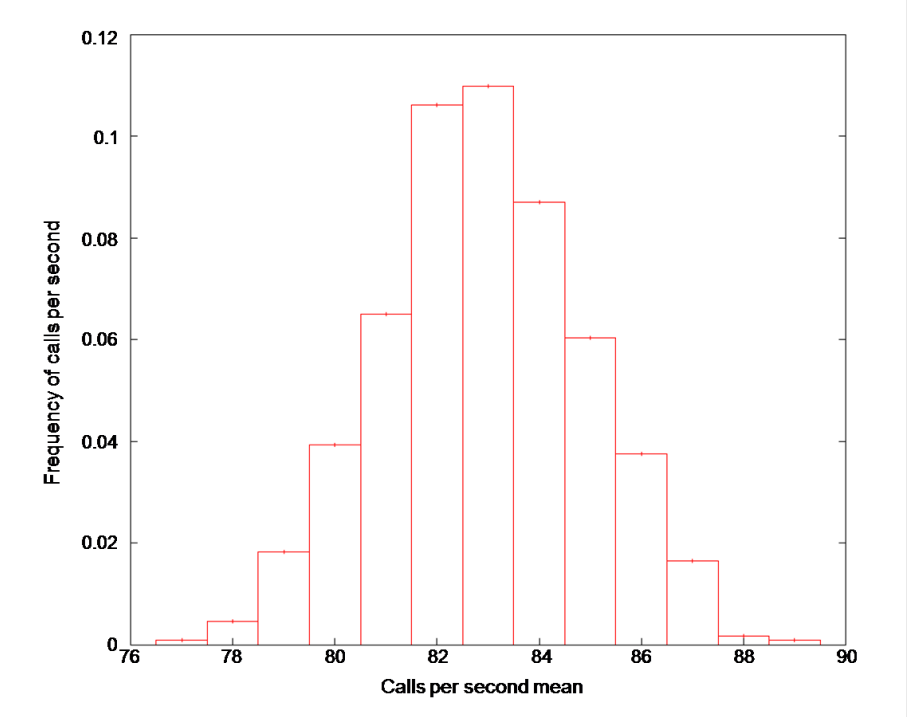

Figure 3.6: Example of Histogram with the Distribution of Scenario Requests

3.4.4 Traffic-Time Profile

A further concept is the traffic-time profile used to describe how the load intensity, i.e., arrival rate of scenarios, changes along the test duration. The traffic-time profile defines the load intensity as a function of elapsed time during a performance test. The function should ensure that sufficient samples are generated on a specific time duration so that the performance metrics can be collected with an appropriate confidence bound.

The traffic-time profile combined with the traffic set guarantee that the workload is composed as a mix of scenarios, where each scenario with a specification and that the duration of the load intensity at a specific level is long enough to collect relevant measurements for performance evaluation.

A common traffic-time profile is the stair-step traffic-time profile which is based on the *stair-step* shape (see Figure 3.7). The width of the stair-step is such chosen to collect sufficient samples at a constant average scenario arrival rate. In the presented methodology only the *stair-step* shape is considered so far.

The traffic-time profile is controlled by a set of test parameters which regulate the behaviour of the test along the execution time, e.g., load generation. As long as the traffic-time profile relates to a particular traffic shape, the parameters can be defined globally for that shape. The general parameters for the stair-step traffic-time profile are:

- *stir-time* - amount of time that a system load is presented to a SUT at the beginning of a test. During this time interval, the database records are said to be "stirred".
- *total number of simulated users* - this describes the total number of simulated users. The

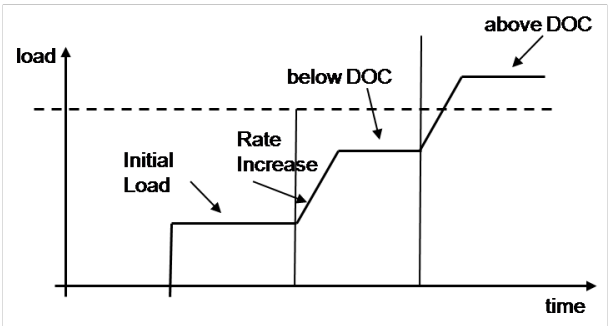

Figure 3.7: Stair-Step Traffic-Time Profile

Table 3.2: Traffic-Time Profile Configuration Example

Stair-Step Traffic-Time Profile Parameters	Value
PX_SimultaneousScenarios	2
PX_TotalProvisionedSubscribers	20000
PX_PercentRegisteredSubscribers	100%
PX_StepNumber	4
PX_StartStepTransientTime	60 sec
PX_EndStepTransientTime	60 sec
PX_StepTime	600 sec
PX_SApSIncreaseAmount	10
PX_SystemLoad	100

SUT has to provide, i.e,. recognise their identities, this number of subscribers during the test execution, but the test system may simulate only a part of them.

- *the percentage of active users* - this parameter describes the average percentage of simulated users that are considered active and used by the TS. For example, if the SUT is required to support one million subscribers, the TS may be requested to simulate only 20% out of this number.

- *scenario increase amount* - the amount of scenario attempts by which the scenario arrival rate is increased.

- *the number of steps* - the number of steps in a performance test as described in the performance test procedure (see Figure 3.7)

- *step duration* - amount of time for a test to be executed with a given system load (a test step) before incrementing the load.

- *start transient time* - an interval at the beginning of a step, during which scenario attempts are not counted. It has the role to make the TS wait until the SUT accommodates to the new load so that the metrics of a previous step are not influenced by the new step.

- *end transient time* - interval at the end of a step, to make the TS wait for the current transactions close. The TS maintains the current load but does not count the metrics for any of the

new created transactions.

- *system load* - initial rate at which scenario attempts arrive.

Table 3.2 illustrates a traffic-time profile for running a performance test with a population of 20000 users. All users are active at the beginning of the test. The initial system load is 100 Scenario Attempts per Second (SAPS) and it is then increased four times with 10 SAPS. The duration of each step is 600 seconds.

3.4.5 Scenario based Performance Metrics

The performance metrics may be defined either for a specific type of scenario or for the whole workload. First the scenario based metrics are discussed.

To validate the DOs established for a particular scenario a set of metrics specific to that scenario can be derived.

Pass/Fail Metrics. The scenario attempts can be sorted into *passed* and *failed* scenarios. The number of failed or passed scenarios represented as percentage of the total attempts is a metric defined for each type of scenario. This metric is usually reported as a time based shape of fails or passes per second and are described by the formula (1) - (4).

(1) $M_{ScenariosPassed} : N \rightarrow [0..100]$

(2) $M_{ScenariosPassed}\ \Delta t = \frac{\#PassedScenarios\ \Delta t}{\#Scenarios\ \Delta t} * 100$

(3) $M_{ScenariosFailed} : N \rightarrow [0..100]$

(4) $M_{ScenariosFailed}\ \Delta t = \frac{\#FailedScenarios\ \Delta t}{\#Scenarios\ \Delta t} * 100$

The metric defined in (3) and (4) computes the number of fails per second but does not take into account which transaction has failed. As long as the interaction flow typically consists of several transactions (see Figure 3.3 in Section 5.4), similar metrics can also be defined at transaction level as presented in formula (5), with the meaning that reports the scenarios which failed on a particular transaction.

(5) $M^{tr}_{ScenariosFailed}\ \Delta t = \frac{\#FailedScenarios^{tr}\ \Delta t}{\#Scenarios\ \Delta t} * 100$

Transmission Latency. Another important metric applied to any scenario is the *transmission latency* which measures the time needed by the SUT to deliver a message from one user to another user. This metric can be computed per second and it is used to compute the average latency of the SUT to process a certain event. It can be defined for any message which transverses the SUT and it may be used to debug the performance of the SUT at transaction level, e.g., help finding which transaction requires the most processing time. The definition of this metric is presented in (6) where Td and Ts represent the time at destination or the time at sending of the message,

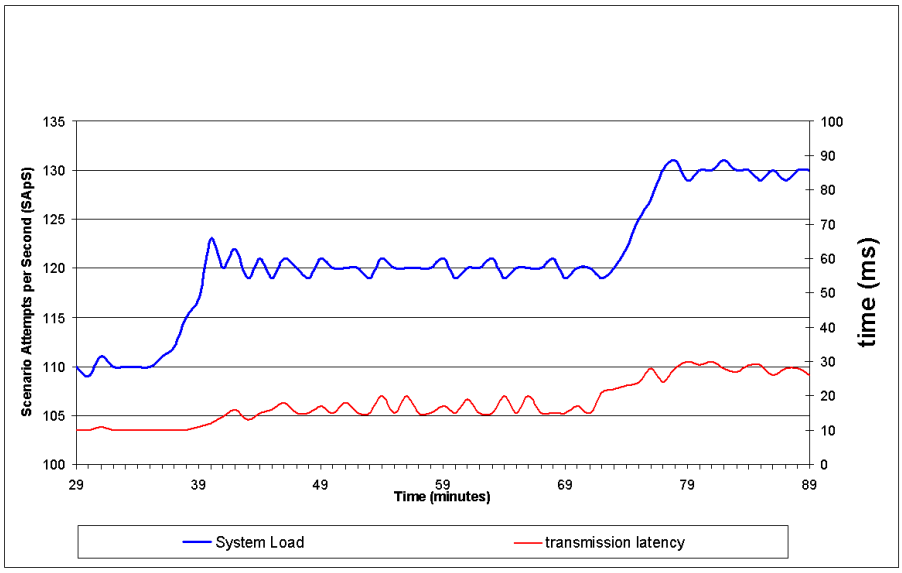

Figure 3.8: Transmission Latency Visualisation Example

respectively. The formula computes the average latency per Δt unit of time out of the number of messages which traverse the SUT in this amount of time.

$$(6)\ M_{latency}\ \Delta t = \frac{\sum_{i=0}^{\#messages}(Td_{message_i} - Ts_{message_i})}{\#messages}\ \Delta t$$

An example of the visualisation of the transmission latency is presented in Figure 3.8 which displays the load intensity as SAPS - upper line - and the latency as time duration in milliseconds - lower line. Both shapes refer to the same X-axis but have different scales. Therefore, the left Y-axis regards the load intensity with ranges from 100 to 136 SAPS while the right Y-axis concerns the latency with ranges from 0 to 100 ms. The duration of the execution is represented on the X-axis. The graph reveals how the latency depends on the load intensity along the test execution time.

Round-Trip Time. The time between sending the request and receiving a response within a transaction is called *round-trip time*. This metric may be used to compute the average time needed to execute a transaction. Similar to the transmission latency, the round-trip time is computed as a function over time. Each second displays the average round-trip time computed among all transactions within that second.

Formula (7) describes this metric. The function $transaction_{duration}(i)$ denotes the time elapsed from sending the first message in the transaction to the last message of the transaction. The average is computed per unit of time Δt out of all transactions created in that time.

$$(7)\ M_{RoundTripTime}\ \Delta t = \frac{\sum_{i=0}^{\#transactions}(transaction_{duration}(i))}{\#transactions}\ \Delta t$$

3.4.6 Global Performance Metrics

Different to scenario based metrics, the global metrics characterise the overall test execution either cumulated for all use-cases or for a particular use-case. They are classified: *resource usage metrics* and *throughput metrics*.

Resource usage metrics. Typical resources that need to be considered include *network bandwidth requirements, CPU cycles, cache, disk access operations, network and memory usage*. These metrics capture the resource consumption of the SUT over time and may help identify *when*, i.e., at which load and after how much time, the SUT runs out of resources or starts failing. The use of multi-core CPUs adds the requirement for CPU monitoring per core. This offers the possibility to monitor how good does the SUT take advantage of processor parallelisation.

Throughput metrics. The throughput metrics are related to the message rates and characterise the quantity of data the SUT is able to process. They are computed globally or per use-case for all scenarios. The following global metrics are introduced:

- *Scenario Attempts per Second (SAPS)* - all observations made for an SUT are related to the applied load. Therefore, the load intensity is a metric to be computed along the execution time. This metric has the name SAPS and is computed as the average rate at which scenarios are attempted. Here, all scenarios - not only the successful ones - are taken into account. By observing this metric as a function of time, it becomes easier to understand how all other metrics evolve over time and how they depend on the load intensity.
- *Simultaneous Scenarios (SIMS)* - another important metric is SIMS which counts how many scenarios are open in each second. The duration of one scenario varies from a few seconds to several minutes, therefore, SIMS metrics is a good indicator of how many open calls can handle the SUT.
- *Inadequately Handled Scenarios (IHS)%* - to compute the error proportion, the IHS% metric has been introduced. This metric is computed as a proportion of Inadequately Handled Scenarios Attempts (IHSA) out of the total number of attempted scenarios. Therefore, the IHS is represented as a percentage, i.e., IHS%. A scenario is counted as failed when either: a) the message flow has not been respected, b) the message flow was stuck (timeout) or c) one of the DOs for the scenario is not met. If one of these situations happens, then the scenario is considered to be an inadequately handled scenario.

3.4.7 Design Objective Capacity Definition

The primary comparison metric reported by the TS is the Design Objective Capacity (DOC) and it represents the largest load an SUT can sustain under specified conditions, e.g., error rate should not exceed a given threshold. The DOC is intended to characterise the overall performance of an SUT. It should not be confused with DOs which describe requirements with respect to delays or error rates of a scenario. This number should serve for comparison among versions of the SUT, different hardware platforms, different products, etc.

This number is defined as the *load intensity* for which the SUT *still can handle the load for certain quality conditions*. Increasing the load intensity would automatically affect the SUT to exceed

the demanded performance requirements, e.g., response times should be below a predefined limit. The execution of a performance test implies that the selected scenarios from various use-cases are executed at the same time. Each started scenario becomes a *scenario attempt*. Any of the scenarios not handled correctly count as IHSA. The IHSAs are used to compute the IHS% metric which is compared with a predefined threshold which is chosen by the test engineer. The last value of the load intensity before IHS% exceeds the threshold is taken as the DOC value for that system.

3.4.8 Performance Test Procedure

A performance test is executed following the selected traffic-time profile[2]. Based on this shape, several test steps with increasing loads are executed with the scope to measure the DOC of the SUT. The test terminates when the threshold for IHSA frequency is reached. This is done by finding a load at which the error rate is below the threshold, another load at which the error rate exceeds the threshold, and by bracketing the DOC between these two loads. A test starts with an underloaded system, which is gradually brought to its DOC, and maintained at that load for a certain time. The time while the system is running at its DOC must be long enough to provide meaningful data and highlight possible performance issues, such as memory leaks and overloaded message queues.

The DOC interval is searched in accordance with the performance test procedure described in Listing 3.1. The test starts with the initial load and it increases the load for a number of steps. Each step is executed for the pre-established duration of time. The results are analysed after each step in order to evaluate whether the DOC was reached or not. If not, a new performance test with an increased load value is executed. After a number of iterations, the situation depicted in Figure 3.7, where the DOC is surrounded between two load values is encountered. However, the DOC taken into account is the lower load value. In order to find the DOC with a finer granularity, that interval can be split into further load values and run another test.

Listing 3.1: Performance Test Procedure

```
1.      configure the system load for an initial load
2.      run the test for TshortStep duration
3.      if the step has IHS<0.1 increase the load with
        SApSIncreaseAmount and go to 2 else go to 4
4.      decrease the load with SApSIncreaseAmount
5.      execute confirmation run for Tconfirmation duration
6.      if the confirmation run has IHS>0.1 go to 4
7.      the DOC is the current load
```

For better confidence in the results, the procedure terminates with a campaign of confirmation runs. Different from the short steps to bracket the DOC value, the confirmation runs are executed for a long enough period of time such as few million of transactions. If during the confirmation run, the SUT fails, then the DOC must be lower than the first estimation and, consequently, a new confirmation run is executed for a lower value. The test finishes when a confirmation run terminates successfully, i.e., the IHS% is bellow the threshold.

[2] In this thesis only the stair-step traffic-time profile is considered.

3.4.9 Performance Test Report

The performance test report collects and visualises all relevant metrics and statistics. A general structure of a performance test report is shown in Figure 3.9. It is a document, with accompanying data files, that provides a full description of an execution of a performance test. The SUT and the TS, as well as their parameters, are described in sufficient detail that another person can replicate the test. The test results should present as intuitively as possible the computed metrics and data sets in the form of charts, graph or other visual format. A simple inspection of this information depicts the behaviour of the SUT over the whole test execution time.

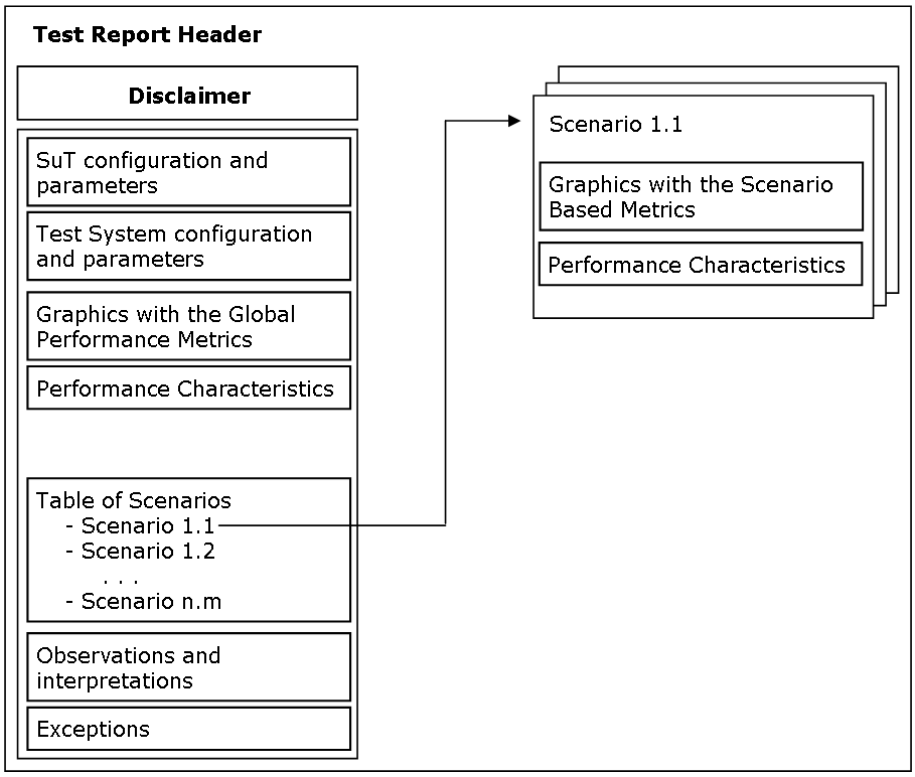

Figure 3.9: Performance Test Report Structure

The performance test report should visualise the global performance metrics and the scenario based metrics, respectively. The metrics are presented in the form of graphs which display the values per unit of time. Additionally, for each metric the performance characteristics should be computed as well. With respect to the scenario based metrics, the test report contains a list of separate reports for each scenario. Each scenario page includes the graphs generated out of the collected metrics for that scenario and their performance characteristics. The types of graphs are detailed in the following subsections.

3.4.9.1 Call Rate and Error Rate

Figure 3.10 provides an example of visualising the SAPS and IHS metrics. The test consists of two load steps. The first line indicates the intensity of the load. Additionally, the points indicating the number of scenarios created in each second can be observed. The dashed line (second line)

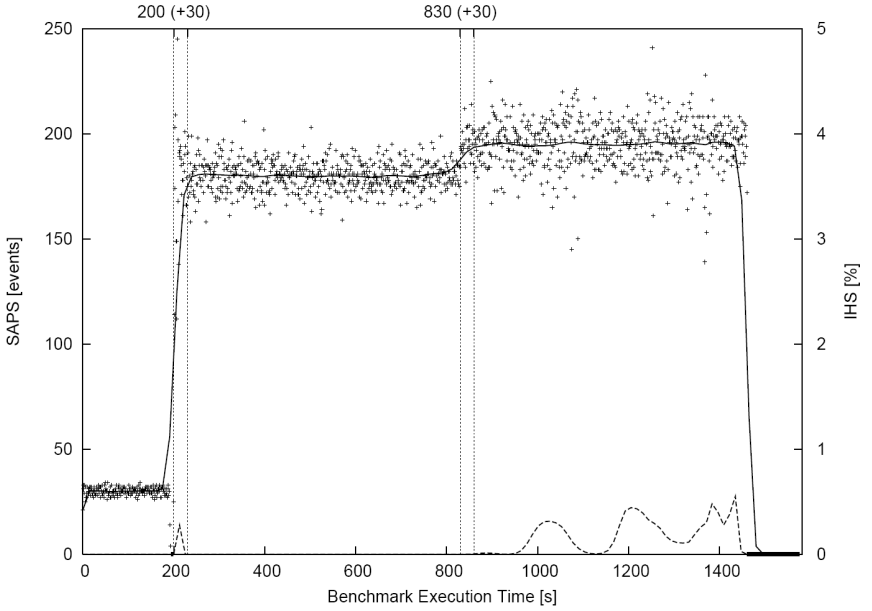

Figure 3.10: Call Rate and Error Rate Visualisation

indicates the IHS as a percentage of fails out of the total number of calls. This graph visualises two metrics in parallel, where one of the metrics is the load intensity itself. The same idea can be applied to all other metrics, and even more than two metrics can be visualised in parallel.

3.4.9.2 Reaction Latency

Figure 3.11 shows the average latency of call establishment. This metric measures the round-trip time between sending a request message until receiving the corresponding response message. The time between the two events denotes the actual computation time required by the SUT to process the request and deliver the response. This graph indicates the dependency between the latency and the load level: during the second load step the latency is obviously higher.

3.4.9.3 Resource Consumption

Besides the protocol related metrics, the TS should monitor resources consumption of the SUT. Figure 3.12 shows the dependency of the CPU system time (dashed-dotted line) and user time (dotted line) on load rate. The demand for this resource depends on the applied system load. Figure 3.13 indicates the memory demand (dashed-dotted line) along the performance test procedure.

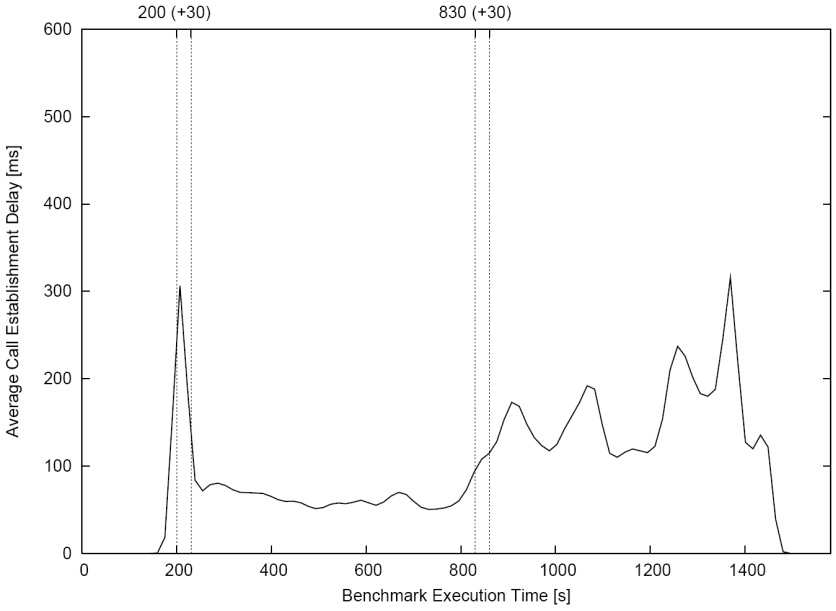

Figure 3.11: System under Test Reaction Latency Visualisation

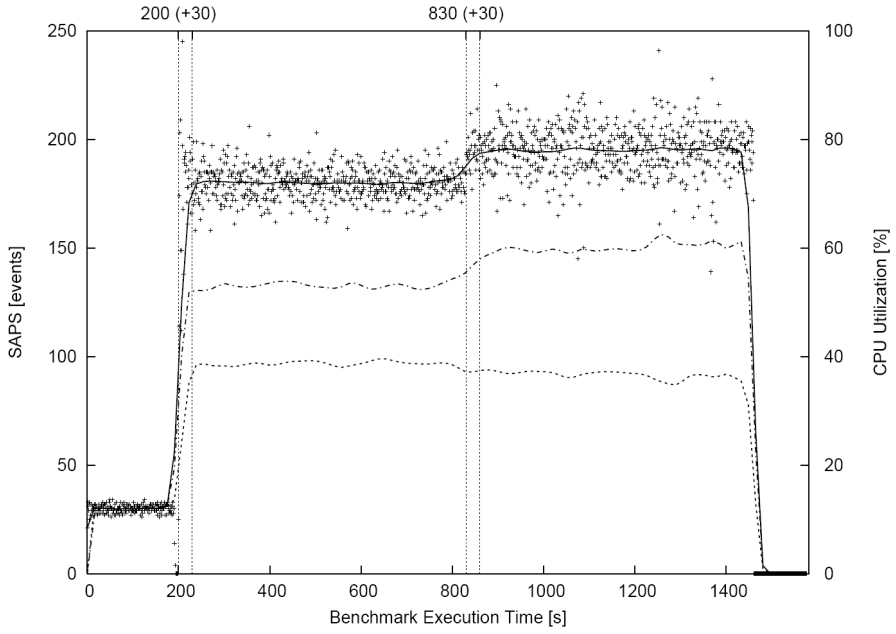

Figure 3.12: Graph Example of CPU Consumption Visualisation

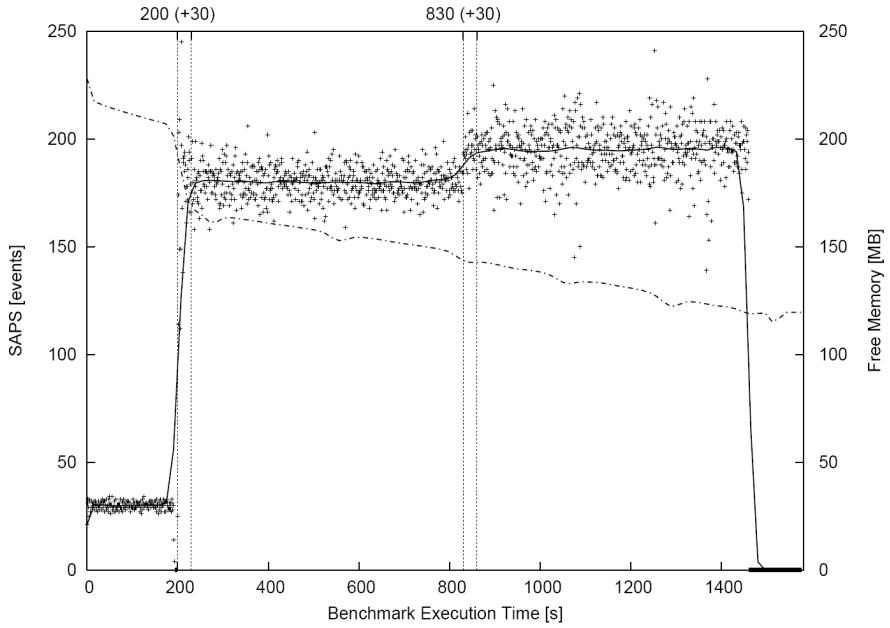

Figure 3.13: Graph Example of Memory Consumption Visualisation

3.5 Workload Implementation Patterns Catalogue

Test patterns are generic, extensible and adaptable test implementations [Bin99]. Test patterns are reusable and, like an analogy of software patterns [Ale77], derived from test methods and test solutions. They are available in the form of design methods [GHJV95], software libraries and/or code generators which provide the test engineer ready to use code.

Although the test implementation languages are very flexible and allow various ways to write a test, very often several architectural design patterns can be recognised in non-functional test implementations [Neu04], [VFS05]. A common practise in software design is pattern-oriented architecture design [SSRB00], [BHS07]. Similarly, architectural patterns can also be defined for performance test development. The intent of this section is to analyse the implementation patterns which may be used in TS realisation.

The overall test behavioural specification is realised as a collection of parallel executing processes. Most operating systems provide features enabling a process to contain multiple threads of control [Tan01]. At this level, the resource contention resolution, scheduling, deadlock avoidance, priority inversion and race conditions are common problems [SGG03].

Within a test platform, the threads are used for testing specific tasks such as load generator, user state handlers but, within the execution platform, also threads dedicated to non-testing tasks, e.g., garbage collection thread in a Java-based platform, may coexist. The focus in this thesis is on patterns related to test behaviour only.

The employed threads fulfil different requirements. They can be grouped into types of actions, e.g., threads for load generation, threads for state handling. The type of a thread may be instantiated for an arbitrary number of times. Additionally, for each type of action, more than one thread can be instantiated in order to increase the parallelism.

The activity of the threads focuses on the interaction with the SUT, e.g., messages preparation, communication, state validation. The behaviour of a thread may consist of simple protocol data interchanges or of more-complex state machines based on sessions with several communication transactions. The complexity consists of data processing instructions, communication instructions, timing conditions. The set of testing related operations and their flow of execution include:

- *user creation* - concerns the creation of a user and its initialisation. At this step, the test process creates a new entry in the users repository and sets the initial status of that user.
- *user termination* - at the termination of a user, the entry in the users repository has to be removed. Additionally, if the communication is transaction based, the associated transactions or timers have to be removed as well.
- *data processing* - different computations appear in a test process , e.g., data preparation, evaluation of SUT answers.
- *encoding* - the data is encoded into the data format of the SUT. This operation only concerns the tools which do not work directly on the raw-data.
- *send* - the operation the TS undertakes to send stimuli to the SUT. This operation also comprehends the underlying communication operations, e.g., socket operations.
- *enqueue* - the operation the TS performs when receiving a message from SUT. Each received message is put in a queue from where it will be processed at a later stage.
- *decoding* - the inverse operation of encoding; a raw message, constituting the returned SUT information, is transformed into a structural entity which can be further investigated by the TS. This step is also characteristic only to those tools which do not work directly on the raw-message, but rather create a structural representation of the message.
- *filtering* - the received messages are filtered upon rules which determine the type of the messages. In practise, the TS has to verify more than one filter. If one of the filters matches the message, the TS executes the actions triggered by that filter.
- *timer timeout* - is the operation to check if an event happens, e.g., the SUT's reaction, synchronisation. A timer is a separate process which issues events for behaviour processes.
- *logging* - is the operation to produce log data when a relevant event happens in the TS.

The interaction between a test behaviour thread and the users' data is depicted in Figure 3.14. The users' data repository consists of users' identity information and the list of active scenarios in which each user is involved. The thread simulates the behaviour of the user which consists, in that example, of the communication operations represented in the figure as *send* and *receive* operations. For simplification, the thread which handles the state machine of a user is named *main thread*. At any receive or send operation, the thread updates the state of the corresponding scenario. When a scenario finishes, the *scenario_id* is removed from the list of active scenarios and the user is made available for a new scenario.

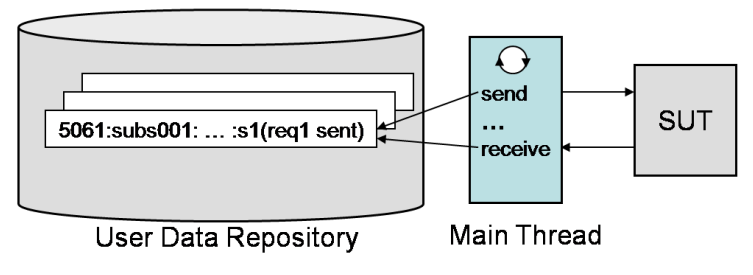

Figure 3.14: User State Handling within a Thread

The following catalogue presents the identified design patterns. They are structured into groups and subgroups which are characterised by their common target.

User State Machine (Section 3.5.1)

- *State Machine with Specific Handler*: The state machine is bound to a single user only.
- *State Machine with Generic Handler*: The state machine can handle in parallel the states of more than one user.

Thread Usage in User Handling (Section 3.5.2)

- *Single User Behaviour per Thread*: One thread handles only one user.
- *Sequential User Behaviours per Thread*: One thread handles more than one user in sequential order.
- *Interleaved User Behaviours per Thread*: One thread handles simultaneously more than one user.

Timers Handling (Section 3.5.3)

- *Timer Handling based on Sleep Operation*: The timer is handled in the main thread using the sleep operation.
- *Timer Handling based on a Separate Timer Thread*: The timers are handled by a separate thread.

Message Sending (Section 3.5.4)

- *Sending in the Main Thread*: The send operation is handled directly by the main thread.
- *Sending with Separate Thread*: The send operation is handled by a separate thread.
  - *Sending with Separate Thread per Request*: The separate sending thread handles only one request; after that it dies.

  - *Sending with Separate Thread per Session*: The separate sending thread handles all requests of a session.
  - *Sending with Thread Pool*: A group of threads, i.e., thread pool, handles all requests of all users.

Message Receiving (Section 3.5.5)

- *Receiving in the Main Thread*: The waiting for received data is performed directly in the main thread.
- *Receiving with Separate Thread*: The waiting for the received data is performed by a separate thread.
  - *Receiving with Separate Thread per Session*: The waiting for the received data is realised in a separate thread.
  - *Receiving with Thread Pool*
    * *Reactor Pattern*: The waiting for the received data is realised by a thread pool which uses the reactor pattern.
    * *Proactor Pattern*: The waiting for the received data is realised by a thread pool which uses the proactor pattern.

Load Control (Section 3.5.6)

- *Single Load Generation*: The load is generated by a single load generator.
- *Multiple Load Generators with Centralised Data*
  - *Push Method*: The load is generated by multiple generators using centralised data. The data is pushed to the load generators.
  - *Pull Method*: The load is generated by multiple generators using centralised data. The data is pulled by the load generators.
- *Multiple Load Generators with Decentralised Data*: The load is generated by multiple load generators which use separate data.

Data Encapsulation (Section 3.5.7)

- *String Buffer Pattern*: The messages are handled as string buffers.
- *Content Reduction*: The messages are represented as structures with minimal content.
- *Structured Representation of Message Content*: The whole content of the messages is represented as structure.
- *Structured Representation of Pointers to Message Content*: The content of the messages is represented as a structure of pointers to locations inside the message.

Population of Users (Section 3.5.8)

- *Single Population*: The population is maintained by a single pool.
- *Clusters of Users*: The population is split into clusters. There is one cluster for each scenario.
- *Minimal Number of Clusters*: The population is split into a minimal number of clusters.

3.5.1 User State Machine Design Patterns

The behaviour of a user is usually implemented as a state machine that stores the status of the user at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change.

Following the definition of the state machine as, for example, the one in [Bin99], the elements of a state machine of the user behaviour in the test behaviour are recognised:

- *an initial state* - the user is initially unregistered.
- *a set of possible input events* - any message received from SUT is an input in the behaviour of a user and may change its current state.
- *a new state may result according to the input* - many types of messages from SUT may change the current status.
- *a set of possible actions or output events that result from a new state* - for most inputs from SUT, a reaction must be taken, e.g., a new message is created and sent back to the SUT as a response.

From the implementation point of view, the user state machine can be accomplished either in a specific way or in a generic way. In the following, these two approaches are discussed.

Pattern: State Machine with Specific Handler (SM_SpecHdl)[3]

Mechanism Illustration

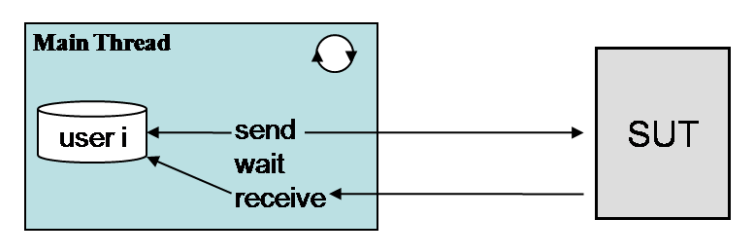

Figure 3.15: Specific Event Handler

Description

The specific handling approach is presented in Figure 3.15. The user state machine is processed entirely by one thread and all data associated to its state machine is stored locally. At receiving or sending of events, the user data is modified locally, the thread does not have to interact with an external repository. This way it avoids synchronisation times with the rest of threads.

Advantages

As far as the implementation of this design model is concerned, it is easy to implement it since the whole logic concerns only one user. The behaviour does not have to perform complex checks to identify which user has to be updated for a given message. The timer events can also be simulated locally within the thread by using sleep-like operations. The thread sleeps for short periods of time until the waiting time expires or a valid event occurs, e.g., a response from SUT is received.

Additionally, this pattern has the advantage that the state machine is implemented very efficiently since at each state only the valid choices are allowed. Everything unexpected is considered invalid and the situation is passed over to an *error handler*. However, this works well only if the behaviour consists of one scenario only. If the behaviour consists of more than one parallel scenario, the state machine may be more complex.

Disadvantages

Unfortunately, for a huge number of users, this pattern is not practical since many threads have to be created [BHS+99]. The most used operating systems (based on Unix or Windows) encounter all serious problems under conditions involving a tremendous number of threads. The more threads are created, the more context switches have to be performed, and the fewer CPU slots a thread is assigned to. Also the synchronisation of many threads may raise a problem since too much time is required. If the test behaviour needs many synchronisation points, it may become too expensive to spend time for synchronisation only [Lea99].

Pattern: State Machine with Generic Handler (SM_GenHdl)

Mechanism Illustration

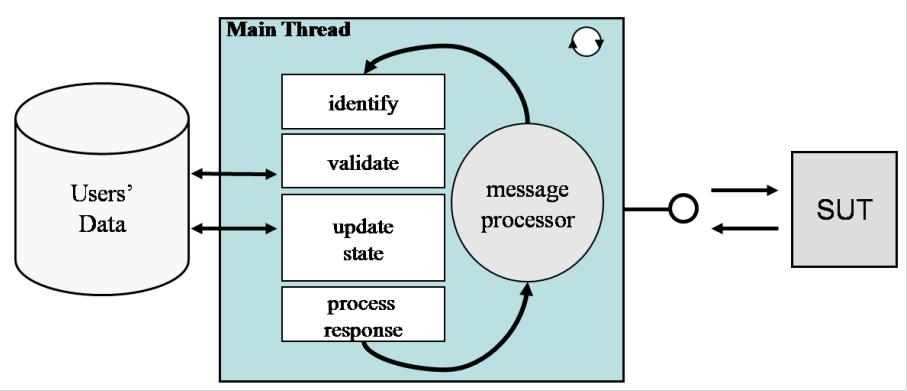

Figure 3.16: Generic Event Handler

Description

The *specific event handler* pattern suffers from performance problems when too many threads are created. Therefore, a better solution would be to create fewer threads by using one thread to handle

more than one user. This way, the platform may scale obviously better than the previous pattern.

This pattern assures the data management globally and the state machine can be rather seen as a message processor. The functionality of this pattern is depicted in Figure 3.16. At any new received message, the message processor identifies the user to which the new message belongs and updates its status in the required way. If the event requires also the creation of a new message which has to be sent back to the SUT, the thread acts accordingly.

Advantages

From a programming point of view, this model is a bit more complicated than the previous one, but the application of some programming conventions eases enough of the technical implementation, e.g., templates, indexing.

This model has the advantage that a handler can be used to process an arbitrary number of users in parallel since it depends on the new received message only. As long as the user data is stored outside the thread, the thread does not control the flow of execution; instead, it executes its actions only when they are triggered by external events.

Another advantage offered by this pattern is that the information of one user may be managed by more than one thread. Since the thread does not manage the user information locally, it might be assigned to handle any arbitrary event. However, all threads have to keep the user information consistent. This concept offers also the possibility of more efficient balancing among the threads since any thread can handle any event.

Disadvantages

A disadvantage in this pattern, is that the handler has to check the type of each new message against all expected message types. This might not be efficient if applied to a large number of types but, by using specialised search methods, such as a tree based search method, the efficiency and optimisations may increase considerably.

3.5.2 Patterns for Thread Usage in User Handling

In the previous section the two models to realise a user state machine have been discussed. Both approaches have advantages and disadvantages. In this section three more patterns regarding the thread design and usage are presented. These patterns are based on the fact that not all users have to be active at the same time. This avoids the existence of inactive threads, by instantiating new threads only when they are needed.

Pattern: User Handler with Single User per Thread (UH_SingleUPT)

Description

The easiest way to implement a user is to create an instance of a thread simulating only that user, i.e. the main thread, as shown in Figure 3.17. This pattern is suitable to the state machine specific handling approach (SM_SpecHdl) but can also be combined with the generic approach. However, if the SM_GenHdl is selected, it is recommended that rather one of the next two patterns is used to handle users.

Advantages

This pattern is easy to apply to user behaviour description. One can also reuse code from functional tests.

Disadvantages

Despite the easiness to write tests using this technique, two main drawbacks exist. Firstly, load control is difficult to realise when the test engineer wants to keep a constant number of parallel users. The test controller needs to continuously control the number of threads acting in parallel and whenever a thread terminates, a new one has to be created. Secondly, the creation, the start and the termination operations applied to threads are very expensive operations with respect to CPU.

Mechanism Illustration

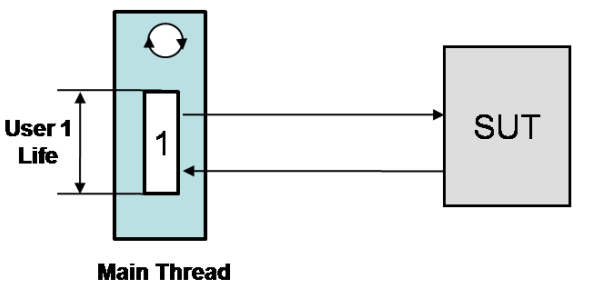

Figure 3.17: Single User per Thread

Pattern: User Handler with Sequential Users per Thread (UH_SeqUPT)

Mechanism Illustration

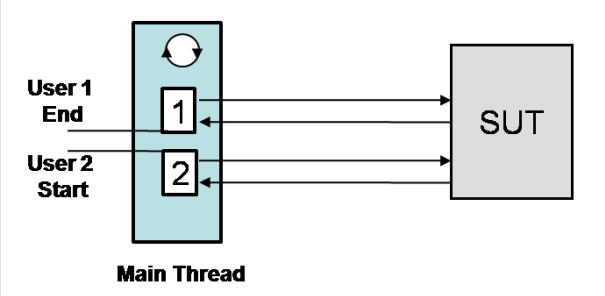

Figure 3.18: Sequence of Users per Thread

Description

Due to the costs to create and destroy threads, it is preferable to *reuse* the threads for further users as soon as other users terminate. This pattern is illustrated in Figure 3.18. This pattern requires that the user identities are stored outside the thread and are loaded into the thread only when the user has to become active. This pattern eliminates the problem of creating and destroying threads. The threads are used like a pool of threads where each thread may take any user identity. During the test, the number of threads may be increased or decreased according to the load intensity. This way, the number of active users is maintained and controlled from within the running threads.

Advantages

The threads are reused to execute new user behaviours in sequential order. This way less CPU will

be spent in order to create and destroy threads.

Disadvantages

This pattern can be combined with the specific strategy to implement a user state machine, as well as with the generic approach. However, if applied to the generic approach, the process does not keep the state locally but in the external users repository.

Another disadvantage is that the number of threads still depends on the load intensity. The higher the load is the bigger the number of threads will be.

Pattern: User Handler with Interleaved Users per Thread (UH_InterleavedUPT)

Description

A better approach to use threads is to interleave user behaviours at the same time on the same thread as shown in Figure 3.19. This pattern may be seen as an extension of the previous pattern with the addition that users are handled at the same time. This way, the thread is able to simulate in parallel an arbitrary number of users.

Advantages

In combination with the generic state handling approach (SM_GenHdl), this pattern is the most flexible and scalable solution. A single thread can handle simultaneously many users. Moreover, there is no need to associate the users to a thread, i.e., different events of one user can be handled arbitrarly by more than one thread.

Disadvantages

If combined with the specific model, the mixture of parallel behaviours on one thread is complicated to specify and most of the time the code loses its readability and becomes difficult to maintain.

Mechanism Illustration

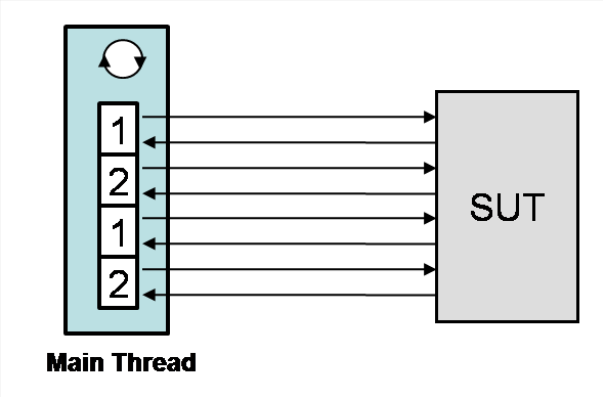

Figure 3.19: Interleaved Users per Thread

3.5.3 Patterns for Timers

All protocols used nowadays in telecommunication include time constraint specifications for the maximal response times. Many protocols include also retransmissions specification at the protocol

level[4]. Additionally, the user interaction with the SUT involves various *user times* such as talking time or ringing time, which in performance tests have to be simulated too.

All these *timing specifications* have been regarded also in the proposed test methodology. The DOs refer to the maximal response times. These values are taken out of the protocol specifications or out of the Service Layer Agreement (SLA). The TS has to validate if these time constraints are fulfilled, otherwise the scenario is considered as *inadequately handled.* If required, the retransmission times are taken directly out of the protocol specifications. They also have to be simulated by the TS. If according to the specification, after a number of retransmissions, the SUT does not react, the scenario is also considered to be inadequately handled. The *user times* are test parameters which are typically randomly distributed around mean values. The mean values are obtained from statistical estimations on the real user behaviours, e.g., the average voice call duration is three minutes.

Two methods to realise timers in test specifications have been identified. Next, these two methods are discussed.

Pattern: Timer with Sleep Operation (T_Sleep)

Mechanism Illustration

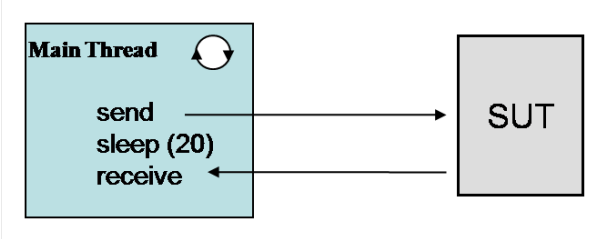

Figure 3.20: Timer Implementation Using Sleep Operation

Description

The sleep operation is a thread functionality which causes the executing thread to sleep for a specified duration. The thread can wait until the time expires or it is notified by an external process to wake up.

The sleep operation can be used to implement the timers used in test behaviours. The operation can be invoked either to wait for an SUT response within a thread, or to validate whether the SUT responds or not. This pattern is illustrated in Figure 3.20.

Advantages

This pattern is simple to apply and the timer concept can be implemented within the same thread as the rest of the test behaviour.

Disadvantages

This approach is very limited due to the thread sleeping time when the thread cannot do something else in parallel. Thus, the thread cannot simulate more than one user since it has to wait for the responses of one user. Similarly, the thread cannot be involved in parallel calls since it has to track the current call. This approach limits the state machine design possibilities to UH_SingleUPT pattern only.

[4]It is not meant the TCP retransmissions, but retransmissions at the upper level of the communication protocol.

Timer with Separate Timer Thread (T_SepTT)

Description

An approach which offers more flexibility implies the use of an external timer thread which manages all timers involved in users' scenarios. The functionality of this pattern is illustrated in Figure 3.21. When an event handler thread comes to the point that a timer has to be started, it asks the timer thread to create a new timer which will notice it back when it expires. The timer thread is provided with the user identity information and the expiration time. The timer thread manages a queue of timers and executes them in their temporal order. When a new timer is created, the timer thread schedules the new event in the right place. Since the timer events are ordered on time base, the timer thread has only to sleep until the next timeout.

When a new timeout occurs, the timer thread notices the events handler thread responsible for the user which created the timer. This can simply happen by sending a timeout event for that user. However, if the user receives in the meantime a valid response from the SUT, the timer thread has to remove the timeout event from the scheduling queue.

Mechanism Illustration

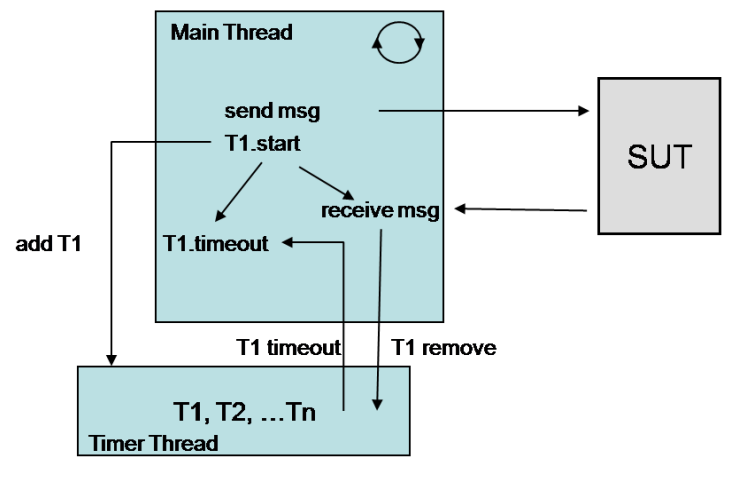

Figure 3.21: Use of a Timer Thread to Control Timing Constraints

Advantages

This approach is compatible with all patterns previously described. It is very flexible since it allows the event handling threads to process events for an arbitrary number of users in parallel or even parallel calls of the same user. Moreover, the timeout events are handled as normal events, which makes the concept more generic. However, for even more flexibility, more than one timer thread can be created.

Disadvantages

One disadvantage is that only one timer thread has to handle all timers created along the test execution. This might cause a problem when too many timers need to be created due to synchronization. However, this issue can be remediated by creating several timer threads.

3.5.4 Messages Sending Patterns

Sending and receiving operations are usually requiring long execution times due to data encoding, queuing and network communication. The threads, which execute them, block until the operation is completed. From this perspective, as argued in the previous patterns, it is not convenient to let a state machine handling thread, i.e., main thread, spend too much time for these operations. Therefore, several implementation patterns are investigated and the problem of gaining more performance from a better design is discussed.

Pattern: Sending with Main Thread (S_MainThread)

Description

The simplest method to realise the call flow of a scenario is to implement it completely as one single thread. That thread takes care of all operations including sending and receiving of events. This pattern is illustrated in Figure 3.22.

This approach is usually used in conformance testing where the test behaviour is a single thread. The thread can send stimuli to SUT or wait for responses from SUT, but never at the same time. When waiting for SUT reactions, technically, the test thread sleeps for short amounts of time and wakes up from time to time to check whether the SUT has replied or not. If the SUT responds, the test thread validates the answer and follows the course of test actions. If the SUT does not react, the test threads can decide to issue a timeout after several wake ups.

Mechanism Illustration

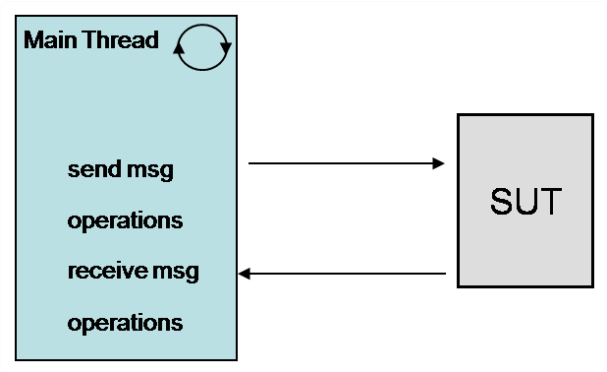

Figure 3.22: Messages Sending in the Main Thread

Advantages

This pattern is simple to use as long as all send operations are handled by the main thread.

Disadvantages

This pattern is not suitable for performance testing for two reasons. Firstly, it is bound to the execution of one scenario, i.e., the thread can only handle messages which belong to one scenario due to the time events it has to wait for. Once the thread has to wait for a response it has to sleep until the thread receives a message or until its waiting time expires. Secondly, in order to simulate a huge number of scenarios, this model is not efficient since each scenario has to be instantiated as a separate thread. For a big number of threads, the TS becomes slow since CPU slots have to be reserved to threads context switches too.

Pattern: Send with Separate Thread per Request (S_SepThreadPerRequest)

Description

A better approach is to use a separate thread for sending operations when these operations consume too much time. This pattern has the advantage that the main thread can work in parallel with the sending thread. The send thread either dies right after sending the message or can also be used as waiting thread to treat the SUT responses. The S_SepThreadPerRequest, illustrated in Figure 3.23, implies that each send request is handled by a separate send thread.

Advantages

This pattern is useful for TSs which simulate multiple users that handle long-duration request/response, such as database queries. The main thread can execute many other operations while the send operation is handled.

Disadvantages

This pattern is less useful for short-duration requests due to the overhead of creating a new thread for each request. It can also consume a large number of OS resources if the TS has to simulate many users that make requests simultaneously.

Mechanism Illustration

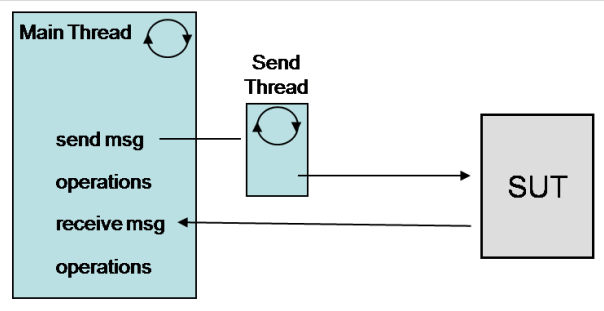

Figure 3.23: Sending with Separate Send Thread per Request

Send with Separate Thread per Session (S_SepThreadPerSession)

Mechanism Illustration

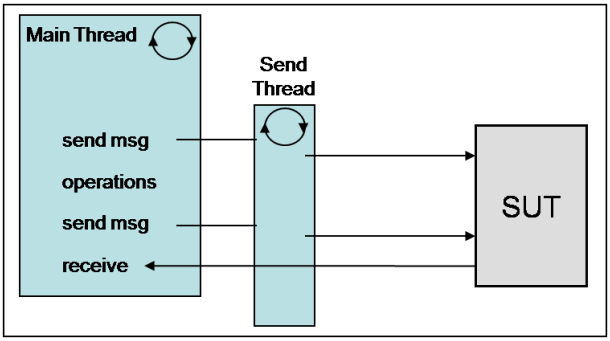

Figure 3.24: Send Thread per Session

Description

The S_SepThreadPerSession is a variation of the S_SepThreadPerRequest that compensates the cost of spawning the thread across multiple requests. It implies that each user simulated by the TS is handled by a separate thread for the whole duration of the session, i.e. the sending operations are handled by the same thread.

Advantages

This pattern is useful for TSs that simulate multiple users that carry on long-duration conversations.

Disadvantages

This pattern is not useful for TSs that simulate users that make only one single request, since this is actually a thread-per-request model.

Pattern: Send with Thread Pool (S_ThreadPool)

Mechanism Illustration

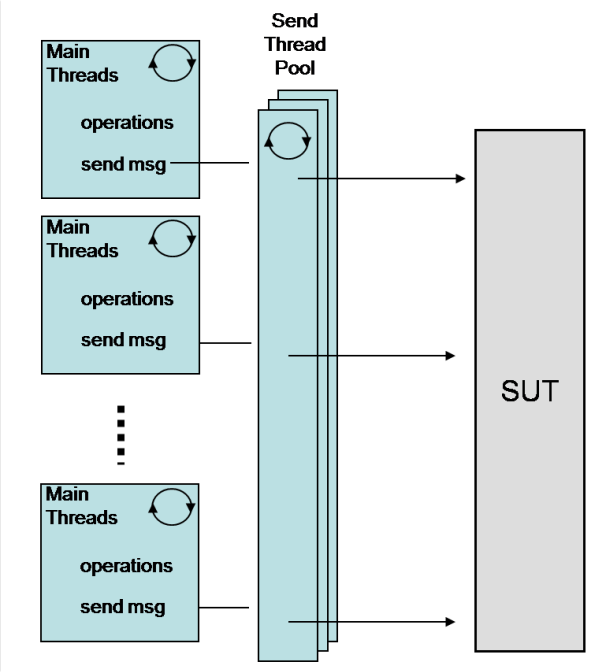

Figure 3.25: Thread Pool for Message Sending

Description

In the S_ThreadPool pattern illustrated in Figure 3.25, a number of N threads are created to perform a number of M tasks, usually organised in a queue. Typically, N is much smaller than M. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available.

Advantages

Having in mind the two patterns described above, the thread pool model can be seen as another variation of thread per-request that also compensates thread creation costs by pre-spawning a pool

of threads. It is useful for TSs that need to limit the number of OS resources they consume. SUT's requests/responses can be handled concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued until a thread becomes available.

The advantage of using a thread pool over creating a new thread for each task is that thread creation and destruction overhead is avoided, which may result in better performance and better system stability. When implementing this pattern, the programmer should ensure thread-safety of the queue.

Disadvantages

One potential problem is how to configure the correct number of threads (N) so that the waiting time for the tasks in the queue is minimal.

3.5.5 Message Receiving Patterns

Similar to sending of messages, there are several patterns to implement the receiving operations. For receiving of messages, the receiver thread creates a message queue in order to read from it whenever new messages are enqueued into it.

Pattern: Receive with Main Thread (R_MainThread)

Mechanism Illustration

This pattern is illustrated in Figure 3.22.

Description

In this pattern, which is traditionally used for functional testing [BJK+05], the thread which implements the scenario call flow may be used also for receiving messages. In this approach, the thread has to stop until a message is received, thus it cannot be used for more than one call flow. At receiving of a new message, the thread validates whether it is the expected one or not.

Advantages

This pattern is simple to apply as long as the receive operation are described in the main test behaviour.

Disadvantages

This approach is a simple one but cannot be used to implement large scale TSs due to the large number of threads required for large numbers of users.

Pattern: Receive with Separate Thread per Session (R_SepThreadPerSession)

Description

For critical tasks it is necessary to return the control to the thread instead of waiting for SUT responses. This way, the test thread can deal with other actions, e.g., other scenario flows. In this situation, a new thread is created and it only waits for SUT responses and notifies the main thread in case where something is received.

Advantages

This approach is needed especially when the first thread has to handle more than one user. The first thread sends a request for one user and instead of waiting for responses from SUT, it creates another request for a further user. The second thread listens to the communication channels for SUT replies.

Disadvantages

This pattern has the inconvenient that, for large number of users, a huge number of receiving threads need to be created.

Mechanism Illustration

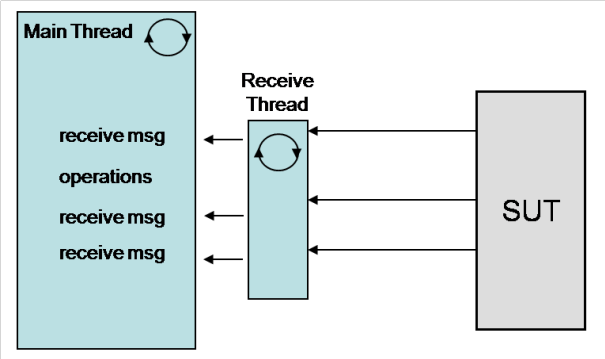

Figure 3.26: Receiver Thread per Session

Pattern: Receive with Thread Pool (R_ThreadPool)

Mechanism Illustration

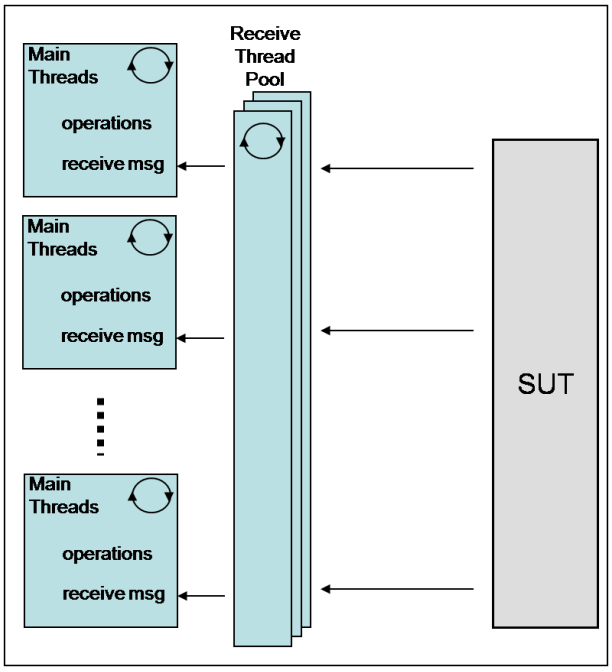

Figure 3.27: Thread Pool for Message Receiving

Description

Depending on the quantity of responses, many different threads might be simultaneously present, complicating the thread management. Since any main behaviour thread requires a second thread

for checking SUT replies, one thread can be shared among several main threads. The shared thread listens to more sockets in parallel and whenever something is received it notifies the corresponding main thread.

This pattern requires an identification mechanism, usually called *event demultiplexor*, between the received messages and the main threads. The event demultiplexor is an object that dispatches I/O events from a limited number of sources to the appropriate event handlers. The developer registers interest in specific events and provides event handlers, or callbacks. The event demultiplexor delivers the requested events to the event handlers.

Two patterns that involve event demultiplexors are called Reactor, i.e., Receive with Thread Pool with Reactor (R_ThreadPool_Reactor) and Proactor, i.e., Receive with Thread Pool with Proactor (R_ThreadPool_Proactor).

The reactor pattern supports the demultiplexing and dispatching of multiple event handlers with synchronous events. This pattern simplifies event-driven applications by integrating the synchronous demultiplexing of events and the dispatching of their corresponding event handlers. The demultiplexor passes this event to the appropriate handler, which is responsible for performing the actual processing. This communication is based on synchronous operations which return control to the caller only after the processing is finished.

The proactor pattern supports the demultiplexing and dispatching of multiple event handlers, which are triggered by asynchronous events. The event demultiplexor initiates asynchronous operations and sends those events forward to the appropriate handlers but does not wait for completion.

Advantages

This pattern uses efficiently the hardware resources by reducing considerably the number of threads.

Disadvantages

This pattern is complicated to realise since it requires an identification mechanism, i.e., *event demultiplexor*, between the received messages and the main threads.

3.5.6 Load Control Patterns

In order to control the load intensity, the TS has to control the number of transactions running in parallel. Such a mechanism is called load control and it is usually implemented as one or more separate processes which interact with each other in order to increase, hold or decrease the number of interactions with the SUT.

Coming back to the general model for a scenario flow, in Figure 3.3, the role of the load controller is to select users and create new calls for those users which executes scenarios as presented in that flow. This implies the sending of the first request, i.e., `Req1`, for each user. This step will bring the user from the **available** state to the **Req1 sent** state, as represented in Figure 3.4. In order to control the intensity of the load, the load controller needs to manage all send events of messages of type `Req1`. For this reason, the load controller is also called *load generator*.

The synchronisation of the parallel threads is realised by passing coordination messages asynchronously or through synchronous remote procedure calls. In general, the load control threads require synchronisation at the start or stop and at increasing or decreasing the level of load.

A general issue of load controlling is the precision of the timestamps of the events. Many perfor-

mance tests require a precise control of the time of each sent event and of the rate the events are sent. Some performance tests even require a precise distribution of the timestamps within a time interval, e.g., Poisson distribution [FMH93].

The load control mechanisms may have centralised data or decentralised data. The centralised data approach implies that the data needed for load generation is centralised in one place, therefore the load generation threads have to acquire the data from a central place. The approaches based on decentralised data eliminate the overhead of synchronisation, so that each load generation thread is capable of deciding how to generate requests.

Pattern: Load Generation with Single Generator (LG_SGen)

Description

The simplest method to realise the load control is to have only one thread to generate the load. A single load generator can control the load by itself taking care of the times of issuing events.

Advantages

This works very well as long this thread can generate the whole load.

Disadvantages

This pattern has the disadvantage that the precision decreases for higher loads.

Mechanism Illustration

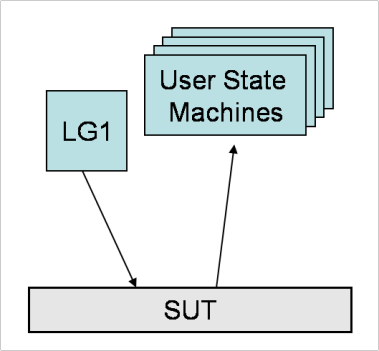

Figure 3.28: Single Load Generator

Pattern: Load Generation with Multiple Generators and Centralised Data (LG_MGenCtrl)

Description

In the centralised approach, the load information is maintained by a central entity called *load controller*. The load controller keeps a global view of the load rate to be realised and controls all timestamps when events have to be created.

The load controller mechanism can work either in *pull mode*, i.e. Load Generation with Multiple Generators and Centralised Data using Pull Method (LG_MGenCtrl_Pull) pattern, in which the user threads ask the controller for timestamps when to send new events, or in *push mode*, i.e. Load Generation with Multiple Generators and Centralised Data using Push Method (LG_MGenCtrl_Push) pattern, in which the controller asks the user threads when to generate new events.

The LG_MGenCtrl_Pull pattern requires that each load generator thread asks the load controller

for the timestamps when to send events. The load controller works in this case as a server which provides an interface for demanding timestamps.

The LG_MGenCtrl_Push implies that each load generator is provided ahead with information when to issue a new event. This requires that each load generator is ready to send events at the demand of the load controller. The load controller works in this case as a dispatcher of timestamps.

Mechanism Illustration

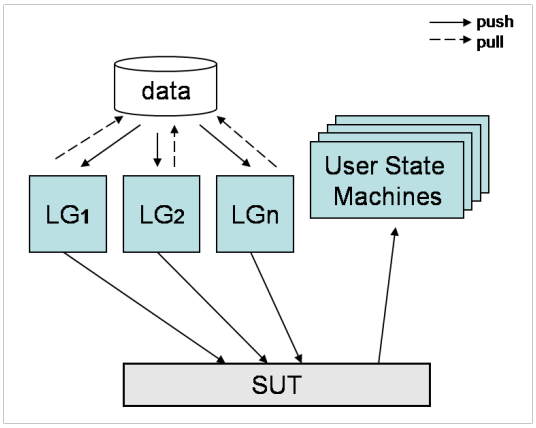

Figure 3.29: Multiple Load Generators with Centralised Data

Advantages

The load generation is distributed to several load controllers which balance the effort of creating requests. This pattern ensures a good time precision.

Disadvantages

The disadvantage of these patterns is that the data is kept centrally, thus, synchronization between the load generator threads is required.

Pattern: Load Generation with Multiple Generators and Decentralised Data (LG_MGenDectrl)

Description

The load control can be realised also in a decentralised fashion, so that each load generation thread acts on its own. The basic idea is to instrument each generation thread for how much load to generate and how it should generate that load.

Advantages

This approach is adequate for distributed execution in order to avoid the communication overhead required by the centralised method.

Disadvantages

One disadvantage of this pattern is that the user data has to be split into smaller clusters so that each load generator can work with its own user data.

Mechanism Illustration

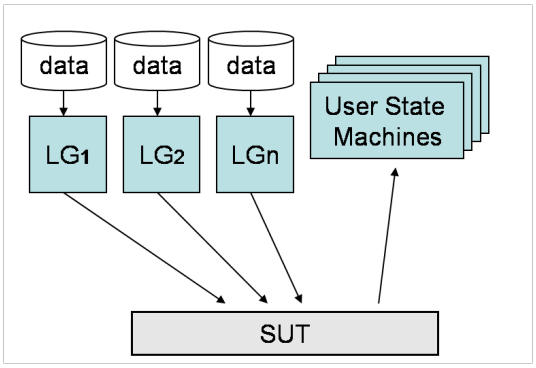

Figure 3.30: Multiple Load Generators with Decentralised Data

3.5.7 Data Encapsulation Patterns

The test operations act upon messages interchanged with the SUT. The content of a message is usually not handled in its raw form but in the form of a structure which provides means to access its information. The mechanism is called data encapsulation. The content of a message is accessed via an interface which provide pointers to the smaller parts of a message.

In this section several strategies to encapsulate and access the content of a message are discussed.

Pattern: Data Representation using String Buffer (D_StringBuffer)

Mechanism Illustration

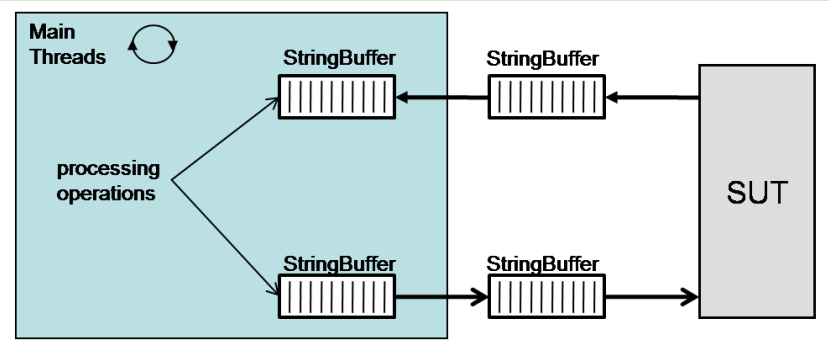

Figure 3.31: Data Representation using a String Buffer

Description

The simplest method to encapsulate message data is to store it into a string buffer. Therefore, this approach is limited to string based protocols, e.g., SIP, diameter. The string buffer allows for easy search and modification operations through regular expressions. Due to its simplicity, this pattern is easy to integrate in any execution environment. It also does not require further encoding or decoding of the content.

Advantages

The string buffer allows for easy search and modification operations through regular expressions.

Disadvantages

Unfortunately, the string processing costs a lot of CPU time. Therefore, the pattern should not be used for big messages. A simple optimisation is to split the message into several strings which are then processed separately.

Pattern: Data Representation with Minimal Structure Content (D_MinStrContent)

Description

This pattern considers that the content is *reduced* to a minimal amount of information. It is based on the idea to extract from the message only the useful information and handle it separately from the rest of the message. When the information has to be used to create a new message it only has to be introduced into the new message. This pattern requires two operations: *decoding*, i.e., extracts the used information from the message and *encoding*, i.e., puts the information into a new message. Figure 3.32 illustrated the mechanism of this pattern.

Advantages

This pattern is applicable to any type of protocol, not only string based. It works efficiently when the used information is small compared to the size of the message.

Disadvantages

This pattern works efficiently only when the extracted information is sufficient to create new messages, i.e. the further messages according to the protocol. Otherwise, when additional information from the original message is needed to create the response, the original message has to be stored as well. This happens when the additional information is not needed by the TS but might be required by the SUT.

Mechanism Illustration

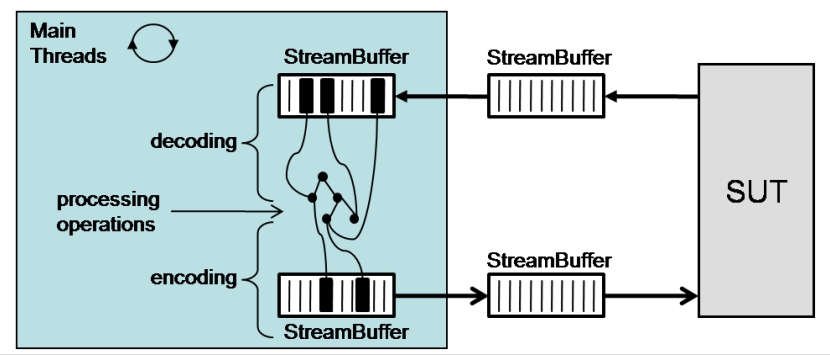

Figure 3.32: Data Representation using a Structure with Minimal Content

Pattern: Data Representation with Complete Structured Content (D_StrContent)

Description

This pattern requires that the content of the message is represented as a tree structure, which is based on the protocol message specification. The information is extracted by a decoder which traverses the whole message and constructs the tree representation. At *send* operation, the tree is

transformed back into a raw-message by traversing each node of the tree.

Advantages

A big advantage, from testing point of view, is that the tree representation eases the access to every piece of information. On top of this the event handler may validate any detail required by the protocol, e.g., test the conformance with the protocol specification.

Disadvantages

The coder and decoder operations may cost a lot of CPU time, therefore it should not be used unless the whole information is required for testing activities. Additionally, the coding operation may need to create lots of short living objects. In a Java based environment, this may cause very dense garbage collections, which make that all threads stops, thus interrupting other important activities.

Mechanism Illustration

This pattern has the same mechanism as D_MinStrContent pattern with the addition that the entire message is decoded into the structured representation.

Pattern: Data Representation as Structure of Pointers (D_StrPointers)

Mechanism Illustration

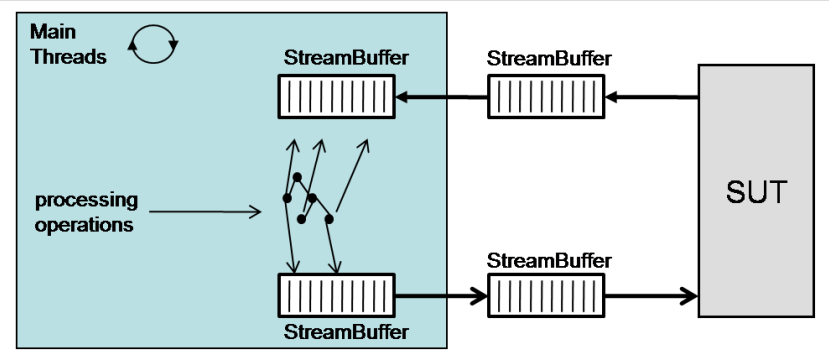

Figure 3.33: Data Representation using a Structure with Pointers to Content Locations

Description

This pattern is an optimisation of the D_StrContent pattern. The idea is to keep in the tree structure only pointers to the locations where the content can be accessed in the raw-message. Figure 3.33 illustrates the mechanism of this pattern. This pattern still requires a CoDec but it can be implemented more efficiently as long as no short living objects are required since the only task is to identify and modify data at given locations in the raw-message.

This pattern can be used even more efficiently when the locations are predefined. In a test workload, where all data is created based on data generation patterns, the locations can be computed in advance only based on the possible content in the message. However, this requires that each field has fixed size, e.g., user number always has seven digits.

Advantages

The main advantage of this pattern is the reduced consumption of CPU and memory for coding and decoding.

Disadvantages

This pattern is difficult to implement for many protocols.

3.5.8 User Pools Patterns

Until this point, the performance test behaviour has been described as the composition of all user behaviours, where each user behaviour is seen as a particular test scenario. However, there still have to be considered the rules to select users and to assign them to a particular type of scenario as long as a user may run any type of scenario.

A user is characterised by its identity and its state. A user can be in "available" state, which implies that the user can be selected to run a new scenario, or another state different than available, while running a scenario. *When a user runs a scenario* it means that all transactions involved in the scenario are parameterised with that user identity.

A simulated user may create calls of different types. Additionally, a user may run simultaneous scenarios, e.g., make a voice call and send a message at the same time. At the creation of a new call, the selection of the user should be arbitrary. The chosen user should be able to call any other user, including himself, e.g., the user can send an instant message to himself. A user may be used several times if the run is long enough. For short runs it is very unlikely that a user is used twice.

The set of users involved in a performance test is called user population and the user population can be split into several user pools. A user pool groups together a set of users which are intended to perform a similar task, e.g., a particular type of scenario, a group of scenarios, a use case.

Any performance test should define and obey a set of rules with respect to the manner in which the users are selected from a users pool. Here is an example of several rules:

- *A user is a state machine able to simulate the complete behaviour of a user equipment*
- *A user may be "callee" or/and "caller"*
- *A user may create more than one call*
- *A user may be reused to create other calls*
- *A user may call randomly any other user*

The "active" users initiating a scenario and reacting to a scenario attempt are selected randomly from one users pool. Since users may be selected arbitrarily, the test logic has to avoid side effects between different scenarios impacting the measurement, as it happens for example in IMS domain when a called party has been unregistered but it has been selected to create a new voice call; thus, the call will fail even though it is not caused by the SUT.

There are several patterns to sort out the users into users pools. These patterns are discussed in the following.

Pattern: Population with Single Pool (P_SinglePool)

Description

The simplest pattern is to use a single pool as presented in Figure 3.34. All scenario instances share this user pool.

Unfortunately, in this approach, it is difficult to avoid conflicts between the different scenarios executed in parallel. For example, for an IMS SUT which requires users to be first registered, the unregistered users for a registration scenario are needed. If a user, which has been already registered, is selected again for registration, this is not a correct selection.

To avoid conflicts, the TS needs more computational resources to perform various checks on the random selected users. If a users does not fulfil the preconditions, then it is abandoned, and another one will be selected.

This pattern influences also the test distribution, since the user states have to be always updated when the call goes into a new state. Assuming that each server keeps a local repository for user information, extra communication between the servers is needed in order to synchronise and keep the local repositories up-to-date.

Mechanism Illustration

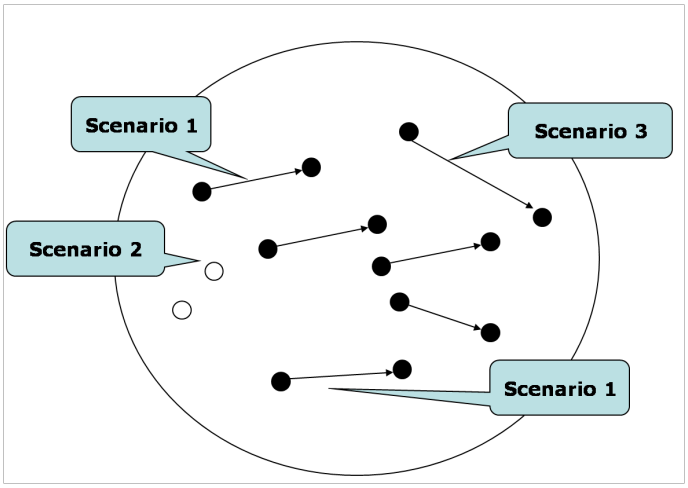

Figure 3.34: Single User Pool Pattern

Advantages

This pattern has the advantage that it is simple to realise.

Disadvantages

This pattern may cause conflicts between scenarios. To avoid these conflicts the TS needs to perform various checks to verify whether the selected users can be used or not in a new scenario.

Pattern: Population with Pool Clusters (P_Clusters)

Description

A better pattern which avoids the conflicts described for single pool pattern defines a separate pool for each scenario as illustrated in Figure 3.35. It is unfortunately less realistic since a user is limited to only one type of scenario along the test duration.

Advantages

This pattern has the advantage that it avoids conflicts between scenarios using the same users.

Disadvantages

This pattern causes a unrealistic behaviour due to the association of users to a type scenario. This

implies that a user associated to one scenario cannot be involved in other types of scenarios than the one it has been associated to.

Mechanism Illustration

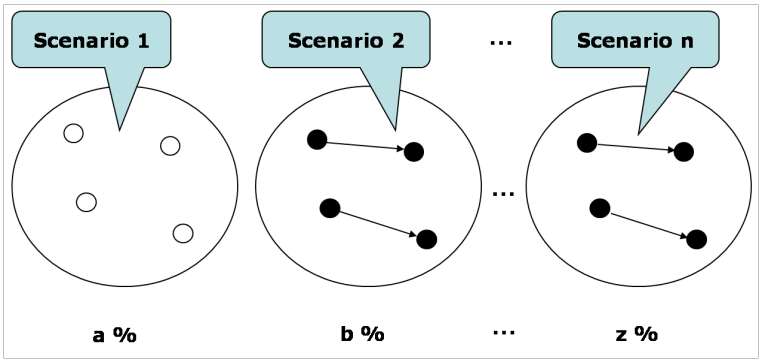

Figure 3.35: User Pools Clusters for Different Scenarios

Pattern: Population with Minimal Number of Pool Clusters (P_MinClusters)

Description

This pattern is based on the combination of the previous two patterns as presented in Figure 3.36. This pattern solves the conflicts between the different scenarios which use the same users, e.g, deregistration versus registration in an IMS network, by creating separate clusters for the users involved in scenarios which might cause conflicts. All other scenarios which do not cause conflicts share the same cluster.

Advantages

This pattern has the advantage that it creates less clusters than P_Clusters pattern.

Disadvantages

This pattern is not realistic if all scenarios should use separate clusters (they may conflict one each other). Additionally, this pattern is more complicate to implement since a special management of the clusters is needed.

Mechanism Illustration

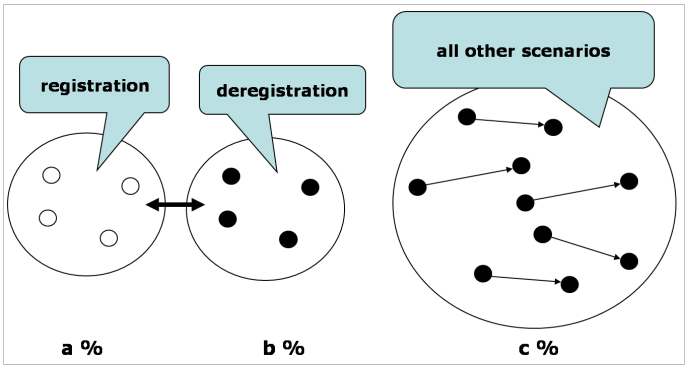

Figure 3.36: Approach with Minimal Number of User Pool Clusters

3.5.9 Pattern Compatibility Table

In the previous sections, several patterns to realise workloads have been introduced and discussed. The patterns have been grouped into categories which depend on the function they have to fulfil, e.g., patterns to realise the event handling. As one can observe, there are patterns from one category which are not compatible with others from another category. The compatibility between patterns is an important aspect which has to be analysed before selecting a certain pattern to realise a workload.

Table 3.3: Pattern Compatibility Table

_	SM_SpecHdl	SM_GenHdl	UH_SingleUPT	UH_SeqUPT	UH_InterleavedUPT	T_Sleep	T_SepTT	S_MainThread	S_SepThreadPerRequest	S_SepThreadPerSession	S_ThreadPool	R_MainThread	R_SepThreadPerSession	R_ThreadPool_Reactor	R_ThreadPool_Proactor	LG_SGen	LG_MGenCtrl_Push	LG_MGenCtrl_Pull	LG_MGenDectrl
SM_SpecHdl	*		+	+		+	+	+	+	+	+	+	+	+	+	+	+	+	+
SM_GenHdl		*	+	+	+		+	+	+	+	+	+	+		+	+	+	+	+
UH_SingleUPT			*			+	+	+	+	+	+	+	+	+	+	+	+	+	+
UH_SeqUPT				*		+	+	+	+	+	+	+	+	+	+	+	+	+	+
UH_InterleavedUPT					*		+	+	+	+	+		+		+	+	+	+	+
T_Sleep						*		+	+	+	+	+	+	+	+	+	+	+	+
T_SepTT							*	+	+	+	+	+	+	+	+	+	+	+	+
S_MainThread								*				+	+	+	+	+	+	+	+
S_SepThreadPerRequest									*			+	+	+	+	+	+	+	+
S_SepThreadPerSession										*		+	+	+	+	+	+	+	+
S_ThreadPool											*	+	+	+	+	+	+	+	+
R_MainThread												*				+	+	+	+
R_SepThreadPerSession													*			+	+	+	+
R_ThreadPool_Reactor														*		+	+	+	+
R_ThreadPool_Proactor															*	+	+	+	+
LG_SGen																*			
LG_MGenCtrl_Push																	*		
LG_MGenCtrl_Pull																		*	
LG_MGenDectrl																			*

In Table 3.3 the compatibility between the presented patterns is analysed. A simple notation is used: the "+" sign marks pairs of patterns which are compatible. Obviously, the patterns from the same category are not compatible with each other, since they cannot be used simultaneously in the same implementation.

Since the load generation patterns consist of separate threads, which do not interfere with the rest of the threads used for user state handlers, timers, senders, receivers, etc., they are compatible with all categories. Therefore, the selection of a load generation scheme does not depend on the way the user behaviour is implemented.

The specific state machine handling pattern cannot be used with the interleaved user handling pattern, while the generic approach for state handling is not compatible neither with the reactor message receiving pattern nor with the sleep based pattern for timers.

Another set of incompatibilities is noticeable for the interleaved user handling pattern. This is not compatible with sleep based timer pattern, receiving in the main thread pattern and reactor based message receiving pattern.

3.5.10 A Selected Execution Model

Throughout the case study presented later in Chapter 5, most of the patterns have been experimented. For any given workload, many design approaches are suitable. However, each pattern has advantages and disadvantages. Some are difficult to implement but work more efficiently, others are simpler to implement but are also less efficient. Though the work presented here may help test engineers to decide upon an architecture, it is, however, out of the scope of this thesis to decide which design or pattern is best to use. The goal is rather to decide upon an example of an execution model and study how performance test workloads can be implemented on top of it.

At the beginning, the TS was based on the *Specific Handler* design pattern (SM_SpecHdl) and on the Single User Behaviour per Thread (UH_SingleUPT). For the receiving of the messages the *Thread per Session* (R_SepThreadPerSession) pattern was used. As far as the data is concerned, for the data encapsulation a structured representation of the message content (D_StrContent) was used. The initial implementation was a simple adaptation of the functional tests to performance testing. This approach was not capable of satisfing the performance needs as required by the project; therefore, further improvements have been investigated.

The first improvement was related to the mechanism used for receiving messages. A thread pool approach for handling the received messages (R_ThreadPool) has been replaced and the performance of the TS increased up to 40%. The increase occurs in fact due to the reduction of the number of receiving threads.

Another milestone in the implementation was the switch from *Single User Behaviour per Thread* (UH_SingleUPT) to *Sequential User Behaviours per Thread* (UH_SeqUPT) and, later on, to the *Interleaved User Behaviours per Thread* pattern (UH_InterleavedUPT). That was necessary mainly because the SUT was capable of supporting more users and the aim was to test the maximal capacity of the SUT. By using this approach the TS increased the number of emulated users from a couple of hundreds up to more than 10.000. Also the throughput performance increased again up to 30% since fewer threads are created than before.

In the early stages only 5 scenarios were supported, but in the end the traffic set consists of 20 different scenarios. This amount of different scenarios made that the Specific Handler pattern was not suitable anymore; thus the adoption the Generic Handler pattern occurred. The Generic Handler proved to be more flexible and easier to maintain and also more suitable for satisfying all the requirements of the workload, e.g., the mix of scenarios, the possibility of a user to handle more than one scenario at the same time.

A final improvement of the test solution is the switch from a structured representation of the message content (D_StrContent) to a structured representation of pointers to message content (D_StrPointers). From the preliminary experiments a remarkable increase of the performance to more than 300% out of the current performance is expected.

As result of various improvements, the overall performance of the TS increased significantly. The resulting selection of patterns is presented in Figure 3.37. Far from being a simplified model, this particular model combines several patterns described in Section 3.5 and limits itself to a subset of characteristics. However, the model is an outcome of various experiments to find the best design in order to satisfy the performance needs in the case study.

An architecture with more than one CPU per host is considered. It results from the fact that most test hardware is nowadays based on multi-CPU architectures. Additionally, the architecture may scale over an arbitrary number of hosts by distributing the functionality and the data among the

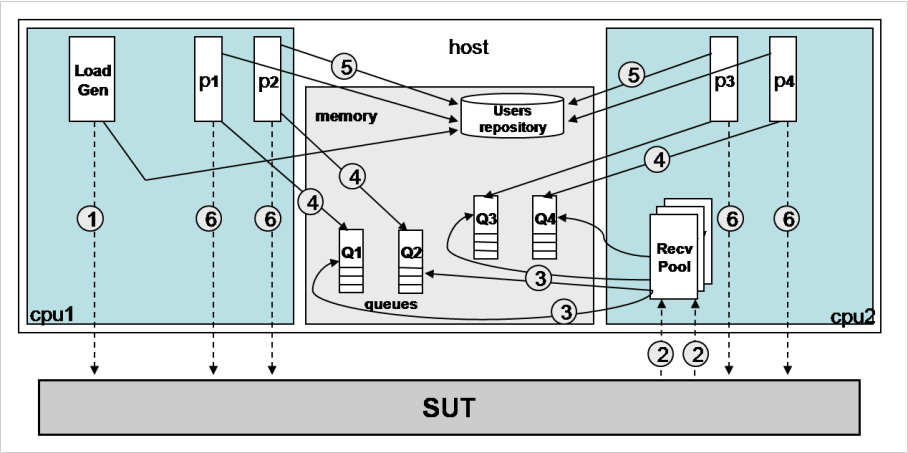

Figure 3.37: A Selected Execution Model

hosts. The execution framework consists of three types of elements:

- *threads* which fulfil various types of tasks, e.g., load generation, state handling, send operations, receive operations, timers operations
- *data repositories* to keep the users' information, e.g., identities, states
- *message queues* to enqueue messages received from SUT or from timer scheduler

The test data consists of user data and traffic set data. The user data comprises of user identity and for each user scenario the scenario status. This data is stored in a user repository created at the beginning of the test. If more than one host is used, the user repository is simply replicated on each host or split into several repositories so that each host has its own data repository.

The traffic data consists of the traffic set, traffic-time profile and set of users to be used during the execution. This data is loaded into the load generators. For a distributed environment, the traffic set is split among the load generators (one per host) so that no collision between users may happen, i.e., each load generator acts on its own set of users.

The load generation is realised by several load generator threads (one per host) which use decentralised data (LG_MGenDectrl pattern). Each load generator has its own data which is created at the beginning of the test. Therefore, each load generator controls only a part of the users. This approach is chosen since it is easy to implement and works for many types of call arriving patterns.

The users are handled by user handler threads in the figure, p1, p2, p3 and p4. On each host, multiple instances of such threads are created. The number of instances is usually selected equal to the number of CPUs on that host such that each thread is executed on a separate CPU. All user handler threads instantiated on a host act on the same user repository in order to update users' status.

In this model an assumption is made that the sending operation does not cost too much CPU. Hence, the user handler threads also execute the sending operations. Therefore, no separate threads for these operations will be created.

On the receive side, a thread pool monitors all communication ports with the SUT. Whenever something is received from SUT, one thread available from the pool reads the message and

enqueues it into the correct queue, i.e., q1, q2, q3, q4. The thread does not have to wait until the message is processed by a user state handler thread, therefore the *proactor* pattern (R_ThreadPool_Proactor) is used.

The received messages are read from the queue by the user handler threads. For each message the handler interacts with the user repository and identify the user responsible for the new message. Then it updates the state accordingly and undertakes the follow-up actions.

The interrupted arrows in Figure 3.37 show the communication paths between the TS and SUT. The continuous arrows indicate the actions within the TS. Each arrow has a label with the sequence number indicating the order in a complete call flow. A call is created by the load generator which selects from the users' repository a valid user. A user is valid when it is not busy and when it can be used for a new scenario. The load generator creates the call by sending the first request to the SUT (1) and updates the status of the scenario in the scenario list of the selected user. The response is captured by the `RsecvPool` (2) which puts the message into a queue (3). Each user handler thread can be used to handle a message, therefore the message can be enqueued arbitrarily in any queue. For optimisation purpose, a load balancing algorithm can be used. Next, the message is read by the corresponding user handler thread (4). The thread processes the state and update the new state in the users' repository (5). If the call flow requires a new message to be sent to the SUT, the thread prepares the new message out of the information available, e.g., the received message and users' repository. The new message is sent to the SUT (6).

In order to simplify the figure, the timer events are not shown. The timer thread is needed to schedule the timer events. Whenever a timeout event occurs, the event is enqueued in one of the receiver queues as any other SUT response. The timer thread can be interrogated at any time by any user handler thread or load generator thread in order to create or remove timers. The timeout events of timers created by load generators are handled also by user handler threads.

3.6 Summary

In this section the composing parts of a TS have been distinguished. The first part describes the method to design suitable workloads and performance tests for multi-service systems. In the second part of the chapter the patterns used in the implementation of performance tests are investigated. These patterns are seen as generic guidelines for how to implement a test relying only on abstract artefacts such as threads, user state machines and data processing operations. Additionally, a selected model for test execution based on the introduced patterns has been also described. The discussion around the model compares different combinations of the patterns as they have been employed within the case study (see Section 5). However, the list of patterns can still be extended if further hardware or OS artefacts such as Hyper-Threaded CPUs [BBDD06], multi CPU architectures [KSG+07] are considered. Additionally, platform, e.g., J2EE [DF03], specific characteristics should be taken into consideration.

Chapter 4

Performance Test Execution

I was born not knowing and have had only a little time to change that here and there.
– Richard Feynman

The methodology introduced in the previous chapter helps a test engineer in designing meaningful performance tests. The next step is to realise those tests into a concrete test execution platform. As shown in Chapter 2 where the requirements for a functional test architecture are discussed, the *test execution* is the TS part which executes the test operations described into a programming language form. The test operations implementation approaches have been presented by means of patterns for the most common operations: state handling, load generation, send/receive, etc. which can be realised by using general data processing elements: threads, buffers, data structures.

This chapter covers the design of the *test harness* for performance test execution and proposes a concrete conception of an execution platform based on the TTCN-3 language. The test harness is the central part of a test execution platform which supplies the functionality to *create, execute and validate* tests [ORLS06]. The elements of the test harness are: stubs, test drivers and test control systems. These components build together the system that starts the SUT, sends messages to it and then evaluates the responses [Bin99].

4.1 Requirements on Test Harness

A test harness is a software just like an application system [Bin99]. Test harness definition includes the specification of requirements, architecture and functional components, interfaces between components and the interface to the user [LyWW03]. This Section regards the requirements and the execution dynamics of the test harness while, later on, in Section 4.6 the architecture and interfaces are presented in detail.

4.1.1 Test Execution Driver Requirements

Test harness can be a library, a tool or a collection of tools to map the abstract test concepts, e.g., users, states, traffic set into operating system elements, e.g., threads, IO, buffers [Bin99]. The requirements for a performance test execution driver are highlighted.

Full automation. First of all, the test execution driver has to be fully automated [FG99], i.e., the test engineer should only need to configure the tests and then wait for the final results. The

test framework should provide the test engineer with all the technical support to compile tests, generate execution scripts and control the execution, e.g., stop, pause, restart anytime during the test.

Distributed execution. Since various types of performance tests are regarded, the performance of the test tool should be very high. It is also expected that the SUT will scale very fast due performance improvements, scaling over more servers, etc. Therefore, the test tool should also be able to scale its overall performance by distributing the test behaviour over several hosts.

Test Reporting. The test reporting capability constitutes an important aspect. During or at the end of the test execution the information about the SUT's performance has to be reported. This can be gathered from the execution traces, therefore, an important requirement for the test driver is the logging provision. However, the logs alone are also not enough. The reporting feature has to provide various graphs and statistics toward performance metrics.

4.1.2 Execution Dynamics

The execution of a test application aims at installation and execution of a logical component topology in a physical computing environment. It comprises the following steps:

1. The code is uploaded on all target test nodes.
2. The execution platform allocates on each node the required start-up hardware resources, e.g., memory.
3. Start of the execution environment and instantiation of parallel processes on particular target nodes.
4. During the execution, the creation and connection of further parallel processes is foreseen. The processes are programmed to execute test operations, interact with the SUT and detect SUT's malfunctions.
5. When the test procedure is done, the execution environment stops the parallel processes, closes the communication channels to SUT, delivers the produced logs to the tester and releases the allocated memory.

4.2 Performance Testing Tools Survey

There are various commercial and open source libraries or tools which can be used for performance testing. Although there are many documents put out by software vendors, very few research surveys of performance test tools are available. Also very few research papers present performance test tools design issues.

In research catalogues like ACM [ACM08], CiteSeer [Cit07], Elsevier [Els07], Springer [Spr07], IeeeXplore [IEE07] only a few publications related to performance test tool design can be found.

In [KPvV00] the Tool Command Language/Toolkit (Tcl/Tk) [WJ03] language is used to develop performance test frameworks. The paper explains the complex requirements of load testing and gives a detailed overview for the extensive use of Tcl/Tk within the system. Similarly, in [Ama98] a Tcl-based system capable of running multiple simultaneous tests in the context of testing an

authentication server is presented. The design and implementation illustrate novel applications of Tcl as well as the rapid development and code re-usability inherent in using Tcl as an application glue language.

Tcl combined with Tk is used often in large companies, small companies, academia or source but when used for testing purposes one has to also care about the technical complexity of the programming language itself instead of targeting the testing activity only. Another automated load generator and performance measurement tool for multi-tier software systems is described in [SA07]. The tool requires a minimal system description for running load testing experiments and it is capable of generating server resource usage profiles, which are required as an input to performance models. The key feature of this tool is the load generator which automatically sets typical parameters that load generator programs need, such as maximum number of users to be emulated, number of users for each experiment, warm-up time, etc.

In [Sta06] the Hammer [Emp08], LoadRunner [Mer07] and eLoad [Emp07] are compared. These tools are restricted by licenses and are very expensive; therefore, the paper also motivates the implementation of an in-hause tool for performance testing. The test tool is based on Visper middleware [SZ02] which is written in Java and provides the primitives, components, and scalable generic services for direct implementation of architectural and domain specific decisions. One of the major design requirements for the framework is transparency to the user. Thus, the distributed middle-ware provides a consistent and uniform view of how to build and organise applications that run on it.

Most available documentation is published either by the tool vendors in the form of Web pages or white papers or by the engineers of other companies as technical papers in one of the on-line technical journals. Some popular on-line technical journals are JavaWord [Jav08], Dev2Dev [BEA07], DevelopersWork [IBM07a] and Software Test and Performance [BZ 07]. These journals usually offer users guides, technical details, examples, experiences or technical opinions. A list of open source performance testing tools is available at [Ope07] while another list including also commercial tools is available at [Fau07]. In [Bur03] analyses a number of load testing tools and identifies the specificity of each application, what are the possibilities and finally points out advantages and drawbacks. Some of these tools are referenced in the next section where a detailed comparison is realised.

Many tools for running performance tests exist and consequently also various approaches to execute the performance tests are applied. Table 4.1 captures several tools and some of the features of interest. These tools are compared in the following subsections based on different criteria.

4.2.1 Domain Applicability

One criteria to characterise the performance testing tools is the domain applicability. Many tools are conceived as specialised tools focusing on a particular type of SUT, e.g., Web applications. Other tools are more general being architected to serve for testing SUTs from multiple domains.

Library tools. A simple approach is to implement the workload in a target language, e.g., C, Java, by using a library. An example of such approach is [SNLD02]. The implementation is a specific workload and only that specific workload can be executed with very little flexibility. This approach lacks of extendability and introduces a limitation degree with respect to usage.

Table 4.1: Performance Testing Tools Comparison

Name	Domain Applicability	Scripting Interface	Workload Distribution	SUT Resources Monitoring
SQS Test Professional [AG08]	any	user scripts (may invoke other programs)	programmed by user, flow control possible, heterogeneous	-
WebLOAD [Ltd07]	Web	Java script, visual GUI	automated, GUI, heterogeneous	yes (MEM, CPU, throughput)
WebPerformance [Web08]	Web	recording of actions done in a Web browser	-	-
LoadRunner [Mer07]	any (offers templates for many protocols)	GUI to edit actions by using predefined templates, script editing possible	programmed by user, homogeneous/ heterogeneous (single OS)	system monitors and diagnostics modules
Rational PerformanceTester [IBM07b]	any	scripting (Rational libraries based)	programmed by user, heterogeneous (many OSs)	capture communication for HTTP, HTTPS, SQL
AppPerfect Load Tester [Cor07]	Web	script recording	automated, heterogeneous (many OSs)	integrated monitoring of target machine's system resources
Embarcadero Extreme Test [Emb07]	database, HTTP	-	-	use health index to appreciate the performance of the DB
Microsoft WAS [Mic07]	Web	GUI/recording	programmed by user, heterogeneous (many OSs)	no
Silk Performer [Bor07]	any	workload editor /importing JUnit scripts	automated (based on an agent health system), heterogeneous	no
QA Load [Com07]	Web server applications	GUI/recording/ scripting	programmed by user, heterogeneous (Windows OS only)	no
eLoad Hammer Suite [Emp07]	Web	scripting/ automated	programmed by user, heterogeneous	no
Apache Jmeter [Apa07]	Web	GUI/recording	programmed by user, heterogeneous (any OS)	no
OpenLoad [Sys07]	Web applications	Web based importing data from Excel tables	programmed by user, heterogeneous	no

Specialized tools. A slightly different approach is to derive an execution platform where the specifics of a workload category can be easily implemented, e.g., workloads for a particular technology such as SIP. This concept provides an excellent framework for implementing workloads. The concept permits extensions and parameterisation of the implementation with further workload characteristics, e.g., new user behaviours. Examples of such tools are: Web Load, Web Performance, Extreme Test or QA Load. A disadvantage is, however, the complexity of the environment itself which is inherited in the workload implementation process. The workload designer has to be aware of the programming language elements inside the framework, e.g., parallel processes, instantiation of actors, management of platform elements.

Generic tools. A better approach is to provide a powerful scripting engine or a GUI which permits writing tests in high-level languages. This concept allows easy extendability of the workload scenarios set but also a more flexible development environment. The test designer does not have to care about the memory allocation, parallel process instantiation, etc. But along with the ease of use, comes also more overhead and resource load on the server and it becomes even more important to look at the load issues. This is the viewpoint taken by tools such as SQS Test Professional, Load Runner, Silk Performer and Rational Performance Tester where the workload is specified really as a test procedure which is passed as execution program to an execution environment.

4.2.2 Scripting Interface

Use of the Programming Language of the underlying Framework. Some frameworks offer only test decorators as programming libraries, i.e., JUnitPerf [Cla07]. These approaches offer good performance capabilities but the test development process is quite difficult as long as the tester has actually to program all technical details such as memory allocation, sockets handling.

Use of an Abstract Notation Language. This approach implies that the tests are written in other language that the language used to develop the framework itself. Some examples are Extensible Markup Language (XML) [JG06], TTCN-3 [Tec08]. Technically, the abstract notation language is meant to simplify the specification by providing only the test specific elements. Such approaches are characterised by simplicity and easy-to-use features. Anyhow, the abstract notation appears as an extra layer above the test execution driver, fact which may cause some performance issues.

Use of a graphical language. The test development can be made even easier by providing a graphical notation to describe tests, e.g., Spirent Protocol Tester [Spi06]. Due simplicity of test development, these approaches might also suffer of performance problems.

Use of a capture/replay tool. The simplest approach is offered by capture/replay tools. Such tools are available for Web site testing, e.g., eLoad [Emp07]. The concept is to capture real life traces of users interactions with the SUT and then multiply and reproduce them. The traces can also be modified or multiplied to simulate higher number of users. These tools are very fast both at development and execution since they do not require interpretation or further development. The disadvantage is that the concept cannot be applied to more complex protocols.

4.2.3 Workload Distribution

The performance of the test system is enhanced through scalability over multiple test nodes. One of the columns in Table 4.1 presents whether the tools can distribute the workload or not. Additionally, for each tool it is detailed how the distribution is realised in terms of heterogeneous versus homogeneous hardware and operating systems. For instance, WebLOAD, Rational Tester,

AppPerfect Load Tester, Silk Performer, Apache Jmeter and OpenLoad support distribution over heterogeneous hardware. LoadRunner and QALoad support distribution only between test nodes running the same operating system. Except for AppPerfect Load Tester, all other tools require that the tester defines manually how the workload should be distributed.

4.2.4 SUT Resource Monitoring

The last column in the comparison table refers the capability of the tools to report SUT resources consumption metrics. Most of the tools do not have this feature even though it is extremely important for performance evaluation of the SUT. WebLOAD and Mercury Load Runner allows for deploying of system monitors and diagnosis modules on SUT side. Embarcadero Extreme Test tool, which targets database testing only, is using a so-called "health index" to appreciate the performance of the SUT.

4.3 Motivation for the TTCN-3 Language

Among the various languages and tools to design performance tests, the TTCN-3 technology has been selected to develop and execute performance tests. The reasons to select this language lay on the facts that TTCN-3, as a standardised test language, is increasingly accepted in the industry as the test specification language. Additionally, various features offered by this language make it a suitable technology to implement performance tests. Some of these features are described in this section.

TTCN-3 is an abstract platform independent test language. Since it hides many technical details, e.g., memory allocation, network communication, data representation, behind the abstract artefacts, such as test components, test behaviours or templates, it is easier to experiment various test specification patterns while implementing performance tests. This allows testers to concentrate on the test specification only, while the complexity of the underlying platform, e.g., operating system, hardware configuration, is left behind the scene. This facilitates the description of complex test scenarios and the handling of very complex data structures.

TTCN-3 is a programming language. Though TTCN-3 is an abstract test notation, it is also a programming language which allows the tester program the tests as any software applications. From an engineering view, the language encapsulate the technical mechanisms under much simpler test specific programming elements.

TTCN-3 is a standardised language. The language has been conceived as a standard test language by a group of academia and industry. Therefore, since many and various requirements have been regarded, the language was conceived to satisfy many testing needs. Thus, the belief that the language might be a good choice for performance testing too.

Test Execution Driver Architecture and API. Within this thesis various implementation details toward the test execution driver are examined. The documentation provided by the ETSI standard for the TTCN-3 language contains a precise programming API for implementing a test execution driver. This opens the possibility to understand any compliant execution tool and use this knowledge at the upper level, i.e., the test specification.

Events handling. Complex distributed test behaviours which involve multiple interacting entities are easier to implement in TTCN-3 than in other languages, e.g., Java, C, since many technical aspects, e.g., memory allocation, socket communication, are hidden by the abstract language artefacts. In general, the performance tests are required to process large numbers of various types of events. The event type specification in TTCN-3 is very flexible while the processing of events can be expressed using sequences, alternatives, and loops of stimuli and responses.

Parallel Test Components. The TS can create multiple test components to perform test behaviours in parallel. The dynamic and concurrent configurations are realised by means of configuration operations, i.e., `create, start/stop, map/unmap, connect/disconnect`. This has a significant impact especially in the case of testing systems like IMS where different user behaviours have to be emulated at the same time.

Concurrent types of behaviours. The behaviours of the test components are defined as functions. A function is used in performance tests to specify client activities within a test scenario. A simulated user may behave in different ways when interacting with the SUT, thus the TS may need different functions implementing different client behaviours.

Inter-component Communication. Another mechanism supported by TTCN-3 is the inter-component communication. This allows the connection of components to each other and the message exchange between them. In performance testing it is used for synchronisation of actions, e.g., all components behaving as clients start together after receiving a synchronisation token, or for collecting statistical information at a central point.

Additionally, several subjective factors influenced the technology selection too.

The TTmex project. The TTmex project [TTm08] between Fraunhofer FOKUS [Fok08] and Testing Technologies [TTe08] had the aim to create a distributed test platform for the TTCN-3 language. The platform opened the possibility to design performance tests with the TTCN-3 language and execute them distributed over several test nodes.

The IMS Benchmarking project. The problem of defining a performance testing methodology originated from the IMS Benchmarking project [FOK07]. The goal of this project was to define a standard performance benchmark for the control plane of an IMS network which supports a rich set of services available to end users. The project required the implementation of a set of performance tests in a standard test notation with the motivation to gain the industry acceptance. The TTCN-3 appeared to be the best choice for these requirements.

4.4 TTCN-3 Related Works

The related work on applying TTCN to performance testing targets either the specification of distributed tests with TTCN-3, or previous versions of it, or concerns the test execution and test distribution over several test nodes.

The first experiments with TTCN applied to performance testing were done with the version 2 of the TTCN language. PerfTTCN [SSR97] is an extension of Tree and Tabular Combined Notation, version 2 (TTCN) with notions of time, measurements and performance. Although it came with interesting concepts like foreground and background load, stochastic streams of data packets, that work has been done for TTCN-2 and has never been extended for TTCN-3.

Some practical guidelines for performance testing with TTCN-3 are given in [DST04]. Using a distributed parallel test environment, the authors evaluate the performance of BRAIN Candidate

Mobility Management Protocol (BCMP). The test system creates and distributes parallel test components which simulate Mobile Node clients. This work highlights some of the major performance issues and test system limits when it is about to generate high loads, e.g., when the load on SUT is increased, the load on the test system also increases. They also conclude that a careful test design and test optimisation is a must in these kinds of tests, meaning that tests have to be designed to work efficiently on the execution platform that accommodates them. Although the work is closely related to the topic of this thesis, it misses the application of behaviour and traffic models which are required for performance testing of multi-service systems.

As far as the test distribution is concerned, which is a significant feature when high loads have to be handled, in [DTS06] a platform which implements the TCI [ETS07c] is presented. It also discusses the possibility of using TCI to realise distributed test execution environments and introduced the idea of using load balancing strategies to optimise the load distribution.

4.5 Workload Realisation with TTCN-3

TTCN-3 offers various concepts to design performance tests. This section renders an introduction into the TTCN-3 language and targets the description of the elements used in the case study to design the performance tests. These elements are test data, state handling, timeout handling and load generation. These concepts are introduced along with small examples of how they can be implemented in TTCN-3. However, the introduction does not aim to be a presentation of the language itself. For a detailed presentation of the language the reader is referred to the standard from ETSI [ETS07a], two introductory papers [Gra00], [GHR+03] and the chapter on TTCN-3 in [BJK+05].

4.5.1 Test Data

The type system of TTCN-3 allows any type of protocol message, service primitive, procedure invocation, exception handling as required for performance test implementation to be expressed. A type can be instantiated as **templates**, **module parameters**, **variables** or **constants**. The templates are concrete message instances used in the communication with the SUT. The module parameters, variables and the constants are used in the test behaviour for implementing the test algorithms. The module parameters allow the parametrisation of tests during their execution. For a better integration with other technologies, TTCN-3 also offers the possibility to import data described in other languages, e.g., Abstract Syntax Notation One (ASN.1), Interface Description Language (IDL), XML [W3C08].

The **type** definition constructs allow the realisation of any data representation pattern presented in Section 3.5.7. To describe basic data types, TTCN-3 provides a number of predefined data types. Most of these data types are similar to basic data types of well-known programming languages, e.g., Java, C. Ordered and unordered structured types are created by using constructs like **record** (ordered structure), **record of** (ordered list), **set** (unordered structure), **set of** (unordered list), **enumerated** and **union**. Their elements can be at their turn basic or structured types.

For synchronous communication, i.e., procedure-based, TTCN-3 offers the possibility to define **procedure signatures**. Signatures are characterised by their **name**, optional **list of parameters**, optional **return value** and optional **list of exceptions**. For a performance test based on synchronous communication, the procedures are also specified in the same way as for conformance

tests.

Templates are data structures used to define message patterns for the data sent or received. Templates can be specified for any message type or procedure signature. They can be parameterised, extended or reused in other template definitions. The declaration of a template contains either a concrete value or a set of possible values or a pattern for possible values.

Listing 4.1: Template for receiving data

```
template Request Request_INVITE_r := {
    transactionId := ?,
    requestLine := {
        method := INVITE_E,
        requestUri := ?,
        sipVersion := *
    },
    msgHeader := ?,
    sdpMessage := *
}
```

The TTCN-3 distinguishes between *send* data and *receive* data. A sent message has to be a complete message, i.e., all fields need to have concrete values and contain all information required by the protocol. The receive data does not have to be complete, i.e., some fields can be specified as "*" to indicate that the value of the field can be missing or as "?" to indicate that the field must contain an arbitrary value. The specification of a receive message is a template of something that is expected to be received. This template mechanism is extremely important for performance testing since it reduces considerably the effort to validate a received message. Usually, in performance tests only a few fields need to be validated. When comparing a received message with a template, the message data shall match one of the possible values defined by the template.

In Listing 4.1, an example of a receiving template is given. The template describes a pattern for an INVITE message in the SIP protocol [IET05]. The message consists of `transactionId`, `requestLine` and `msgHeader` fields. When a new message is received, the TS has only to identify the type of the message encoded in the `requestLine.method` field. The values of the rest of the fields do not matter at message identification step. Once the message has been identified, the TS has to use the information within it. Some fields are required, others are not. To specify this, the question mark "?" is used to say that a field must contain an arbitrary value. The wildcard "*" marks that the TS does not expect any value or a specific value for that field.

4.5.2 Event Handling

An event handler processes events received from the SUT and executes appropriate actions according to the scenario message flow. The event processing starts with the *identification* of the `userid` for which the message is received. This information is extracted from the protocol data embedded in the message. Once the user is identified, the handler evaluates the current state of that user and *validates* if the new message corresponds to a valid state, otherwise the transaction is considered an IHSA. Next, the user state is *updated* in the local user information database. If the received message, requires follow-up actions on the TS side, new messages are created and sent to the SUT. When receiving or sending any message, a log event is generated with precise time-stamp for the evaluation of the SUT latency.

In Section 3.5.1, the specific and the generic event handling patterns have been introduced. Both of them are implementable in the TTCN-3 language.

The specific handling can be implemented as a TTCN-3 function which executes on a component. In Listing 4.2, a simple example for the specific handling mechanism is provided. The function gets a `userid` as start parameter to personalise it to a particular user. At the beginning of the function several variables are defined. As long as the function implements a specific user behaviour, the state variables, e.g., state, also have to be defined. Other variables are defined globally for the whole function. The events are received by port `p` and are handled by an **alt** block. Each event type is caught by a receive statement which is parameterised with a **receive** template. The template must contain matching constraints for the `userid` given as parameter to the function. This is simple to achieve by parameterising the template with the `userid`.

Listing 4.2: Specific Event Handling

```
function user(id userid) runs on UserComponent {

        var stateType state := state1;
        var requestType req;
        var responseType resp;
        ...........
        alt {
                [state == state1] p.receive(requestType1(userid)) -> value req
                {
                        if(match(req.field1, expectedValue) {
                                state := state2;
                                // other actions
                                resp := responseTypeTemplate;
                                resp.field2 := req.field2;
                                resp.field3 := req.field3 + 1;
                                p.send(resp);
                                repeat;
                        }
                        else {
                                state = failureState;
                                makeAvailable(userid);
                                stop;
                        }
                }
                [state == state2] p.receive(requestType2(userid))
                {
                        state := finalState;
                        makeAvailable(userid);
                        stop;
                }
                ...........
                // other states

                [] p.receive {
                        state = failureState;
                        makeAvailable(userid);
                        stop;
                }
        } // end alt
} // end function
```

Once a message matches an alternative, the test takes the appropriate actions. In the example, three possible branches are defined. The first one defines the handle of an event of type `requestType1`. An event of this type can occur only when the user is in `state1`. This condition is specified in the guard of any branch. Next, the event is stored into the `req` variable for further use. In the state

handling, particular fields of the received message can be inspected by using the **match** function. If the received value matches a valid state for the user, the state is updated to the `state2` and a response message is prepared and sent back to the SUT. At the preparation of the response message, the data from the request stored in `req` variable is reused. The **repeat** statement makes the **alt** construct go back to the top and wait for new messages from the SUT. For a non-valid message, the TS has to set the state of the user to `failureState` and stop the component.

In the case of receiving a final message, according to the protocol, i.e., second alternative is matched, the TS sets the state to `finalState`, makes the user available for other calls and stops the component. When an unexpected message is received, e.g., last alternative, the state of the user is set to `failureState`, the user is made available and the component is stopped. The TS may log anytime the state of each user for performance statistics post-processing.

Listing 4.3 provides an example of a generic handler in TTCN-3. It basically rewrites the example from Listing 4.2 in a generic way. The generality reflects in the fact that the function can be used now to handle concurrently the events of several users in parallel. To achieve that, several changes have to be performed on the TTCN-3 snapshot presented in the listing. In the first example, the component has one port which is used for the communication with the SUT exactly for one user. In the second example, the same port is used for all users simulated by the component. This is possible thanks to the **address** type available in TTCN-3. This type allows an address of an SUT entity to be defined which will be connected to a TTCN-3 port. Technically, this allows a component to use a single port for any number of users, where each user is characterised by a unique address. The address is used at sending of messages by putting the **to address** statement after the send statement. This concept optimises the number of ports which require a lot of resources (see the patterns for message receiving in Section 3.5.5).

Listing 4.3: Generic Event Handling

```
function userHandler() runs on UserComponent {

        var requestType req;
        var responseType resp;
        ...........
        alt {
                [] p.receive(requestType1) -> value req
                {
                        if(not validState(req.userid, req)) {
                                setState(req.userid, failureState);
                                makeAvailable(req.userid);
                                repeat;
                        }

                        if(match(req.field1, expectedValue)) {
                                setState(req.userid, state2);
                                resp := responseTypeTemplate;
                                resp.field2 := req.field2;
                                resp.field3 := req.field3 + 1;
                                p.send(resp) to address addressOf(req.userid);
                                repeat;
                        }
                        else {
                                setState(req.userid, failureState);
                                makeAvailable(userId);
                                repeat;
                        }
```

```
                }
                [] p.receive(requestType2) -> value req
                {
                        if(not validState(req.userid, req)) {
                                setState(req.userid, failureState);
                                makeAvailable(req.userid);
                                repeat;
                        }

                        setState(req.userid, finalState);
                        makeAvailable(req.userid);
                        repeat;
                }
                ..........
                // other states

                [] p.receive {
                        setState(req.userid, failureState);
                        makeAvailable(userid);
                        repeat;
                }

        } // end alt
} // end function
```

The next change is to drop the conditions in the guards of the alternatives since now any event can be received. Instead, the `validState()` function is introduced so as to check the state of a user in a repository, e.g., a data structure. The validation is realised upon the `userid` information extracted from the received message and the message itself. If the new state created by the received message is not a valid one, the TS considers the current call as fail and makes the user available for a new call. For a valid state the behaviour looks the same as in the previous example. The state is updated by the `setState()` function.

Further on, the receive templates are now generic templates instead of parameterised templates per `userid`. These templates are not so restrictive as the ones used for the specific pattern but one can compensate by introducing more checks through **match** conditions.

In the previous example, the termination of the behaviour of a user was equivalent to the termination of the component. The last modification is to add to each branch the **repeat** statement since the **alt** statement has to be reused for further messages.

For simplification, the fact that a user can be used in more than one scenarios is not taken into account. To achieve that, both examples should be modified such the `callid`, embedded in the received messages, is processed too. Thus, any state change refers to a particular user and to a particular `callid`.

4.5.3 Data Repositories

Besides the messages exchanged with the SUT, other types data have to be managed by the TS as well. They concern the user information needed to create calls and the information needed to track the status of the calls. The referred data is stored in a repository which can be accessed from any test behaviour.

A simple API to manage the repository is presented in Listing 4.4. It consists of very simple op-

Listing 4.5: Simple Data Repository implemented in TTCN-3

```
type record of integer statesList;
type record of boolean availableList;

type record repositoryType {
        statesList state;
        availableList available;
}

template repositoryType repository := {
        state := {0, 0, 0},
        available := {true, true, true}
}
```

Listing 4.6: Repository Access API through external functions

```
external getAvailable() return userid;
external setState(userid u, state s);
external validateState(userid u, newstate ns) return boolean;
external makeAvailable(userid u);
```

erations to extract or update the information. Typical operations are `getAvailable()` to get an available user to create new call and `makeAvailable()` to release a user. For state management, `setState()` serves to set a new state while `validateState()` is meant to check whether a new state to get into is valid or not. However, the API can be extended to many needs but the principle remains the same.

Listing 4.4: Repository Access API

```
getAvailable() return userid;
setState(userid u, state s);
validateState(userid u, newstate ns) return boolean;
makeAvailable(userid u);
```

Unfortunately, the TTCN-3 language does not offer data management containers, e.g., Queues, Hashes, like, for example, Java does [SM08]. This is a point where TTCN-3 should be improved in the future. One solution is to implement these structures into a TTCN-3 library by using data types provided by TTCN-3 such as `recordof, array`, etc. In Listing 4.5 a simple data repository is provided. However, this implementation might not be as efficient as that provided by the execution platform itself. To access execution platform functionality, TTCN-3 offers the external function mechanism which permits the implementation of pieces of functionality outside of the test specification. The example in Listing 4.6 modifies the API so that the repository access functions are external.

As a remark, in the IMS Benchmarking case study (see Chapter 5), both approaches have been experimented (1) using a pure TTCN-3 repository and (2) using an external repository. The second solution proved to be far more efficient. The root cause lies in the way the data types from TTCN-3 are mapped down to the execution environment. For instance, using the TTCN-3 mapping to Java language [ETS07b], a simple integer is not mapped to a Java [SM08] `int` basic type but to an object. This causes that a repository, based on integer type, implemented in TTCN-3, occupies much more memory than one implemented purely in Java. Also for a simple loop, e.g., `for(var`

Listing 4.7: Performance Test Configuration

```
testcase performanceTest() runs on LoadController system System {

        var default d := activate (defaultLoadController());

        // create EventHandlers
        var EventHandler eventHandler[NR_OF_COMPONENTS];
        for (var integer i := 0; i < NR_OF_COMPONENTS; i := i + 1) {
                // create i-th component
                eventHandler[i] := EventHandler.create;
                connect (self:p2EventHandler[i],
                 eventHandler[i]: p2LoadController);
                eventHandler[i].start (userStateHandler());
                TSync.start; p2EventHandler[i].receive(SYNC); TSync.stop;
        }

        // sender component
        var EventSender eventSender;
        eventSender := EventSender.create;
        connect (self:p2EventSender, eventSender: p2LoadController);
        eventSender.start (userMainHandler());
```

`integer i:=0, i<length, i:=i+1)`, using `i` of type integer, actually the test environment has to do all these operations on objects instead of simple integers.

4.5.4 User Handlers

In TTCN-3, the test component is the building block to be used in order to simulate concurrent user behaviours. The specification of all test components, ports, connections and TS interface involved in a test case is called *test configuration*. Every test case has one Main Test Component (MTC) which is the component on which the behaviour of the test case is executed. The MTC is created automatically by the TS at the start of the test case execution. The other test components defined for the test case are called Parallel Test Component (PCO) and are created dynamically during the execution of the test case. The interface to communicate with it is the Abstract Test System Interface (system).

Parallelism is realised in TTCN-3 through PTCs. The components are handled by configuration operations such as **create, connect, disconnect, start, stop** which are mapped to concrete operations in the execution environment. The configuration operations are called from test behaviours, e.g., **test cases, functions, control**. In Listing 4.7 an example of a test configuration is given. The `performanceTest` testcase is executed on the `LoadController` component which is in this case the MTC component. The testcase creates a number of `EventHandler` components and connects them to the `LoadController`. Each `EventHandler` connects its `p2LoadController` port to the `p2EventHandler` port of the `LoadController`. Each `EventHandler` component starts a `userStateHandler` function. The successful start of the this function is verified by starting a `TSync` timer. If, before the timeout of `TSync`, a `SYNC` message is received then the component has started correctly and the `LoadController` can go to the next component. A different type of component, i.e., `EventSender`, is created as soon as all `EventHandler` components are up and running. The `EventSender` executes the `userMainHandler` whose main task is to create calls which will be handled by the `EventHandlers`.

Listing 4.8: Message based synchronisation

```
// SYNC from EventSender that the load generator is ready
// or it could not initialise
TSync.start;
alt {
        [] p2EventSender.receive(SYNC) {
                TSync.stop;
        }
        [] p2EventSender.receive {
                log("fail during preamble");
                setverdict(inconc);

                log("exiting ...");
                stop;
        }
        TSync.timeout {
                log("timeout during preamble");
                setverdict(inconc);

                log("exiting ...");
                stop;
        }
}

// SYNC to EventSender to start the load generator
p2EventSender.send(SYNC);
```

Another important mechanism provided by TTCN-3 is the inter-component communication which allows connecting components to each other and transmitting messages between them. This mechanism is used in performance testing for synchronisation of actions, e.g., all components behaving as event handlers start together after receiving a synchronisation token. Listing 4.8 provides a synchronisation example between the `LoadController` and the `EventSender` components. The `LoadController` wants to know whether the `EventSender` starts successfully. To achieve this, the `LoadController` starts a `TSync` timer and waits until a) the `EventSender` responds or b) the `TSync` expires. In the later case the `LoadController` has to stop the test since the `EventSender` did not start correctly. The possible responses of the `EventSender` are handled by an alt block. If the `EventSender` responds with a `SYNC` message, then the `TSync` is stopped and the `LoadController` sends another `SYNC` message to the `EventSender` in order to start the test. If a different message is receive, then the `LoadController` concludes that the `EventSender` did not start successfully and it decides to stop the test.

The behaviour of a test component is defined by a function. A function is used to specify user activities within a test scenario, i.e., user state machine. A user may behave in different ways when interacting with the SUT. Therefore, the test system may have different functions emulating different behaviours or only one function which takes care of all possible events.

4.5.5 Traffic Set, Traffic-Time Profile and Load Generation

The traffic set and traffic-time profile contain parameters to control the load generation. Examples of traffic set and traffic-time profile have been shown in Section 3.3. All these parameters are defined as module parameters and are used in the load control.

The load control is a special process which is implemented as an `EventSender` component and is responsible for call creation. A new call is created by selecting 1) one scenario for the call, 2) one or more available users [1] and 3) a time when the first event from the call should be sent to the SUT. The scenario is selected according to the traffic set parameters which describe the proportions of scenarios of each type along the execution. The sending time is derived from the traffic profile which contains parameters to describe the load intensity and the call arrival rate, e.g., Poisson.

4.5.6 Timers

Timers are a further essential feature in the development of performance tests with TTCN-3 in order to evaluate the performance of the SUT. The operations with timers are **start, stop**, to start or stop a timer, **read**, to read the elapsed time, **running**, to check if the timer is running and **timeout**, to check if timeout event occurred. The start command may be used with a parameter which indicates the duration for which the timer will be running. If started without parameter, the default value specified at declaration is used.

In conformance tests, timers are defined on test components and they are used in the test behaviour to measure the time between sending a stimuli and the SUT response. If the SUT answer does not come in a predefined period of time, the fail rate statistics should be correspondingly updated.

Listing 4.9: Timer in a specific state machine

```
timer tWait := 0.2;
alt {
        [state == state1] p.receive(requestType1(userid)) -> value req
        {
                ...........
                p.send(secondMessage);
                tWait.start;
                state := state2;
                ...........
        }
        [state == state2] tWait.timeout
        {
                // the call has failed
        }
        ...........
}
```

For performance tests, the timer mechanism works almost the same. Two patterns to implement a state machine namely the specific pattern and the generic pattern have been discussed. For the specific pattern, the use of a timer is similar to conformance testing. The function running the state machine starts a timer and the alt block handles potential timeout events. Listing 4.9 offers an example of how a timer can be used in the test specification. The `tWait` timer is started when the user received `requestType1` and sends a further message, i.e., `secondMessage`. The time until a response from the SUT should be received is 0.2 seconds.

Unfortunately, with respect to the realisation of the generic pattern for state machine, TTCN-3 is not offering an appropriate mechanism to handle an array of timers. This feature is needed for performance testing since each user needs its own timer. Listing 4.10 offers an example with a modified syntax of the TTCN-3 language, which allows using of arrays of timers. `tWait` is

[1] The number of users depends on the type of the scenario.

an array of timers. Whenever a timer is needed, one of the `tWait` timers is used. The handler of `requestType1` event starts the timer `tWait` with index 0. The timeout events of the timer are handled as usual by using `tWait.timeout` with the addition of → **index i** which helps determining the index in the `tWait` array.

Listing 4.10: Timer in a generic state machine

```
var integer i;
timer tWait[200];
alt {
        [] p.receive(requestType1()) -> value req
        {
                ...........
                p.send(secondMessage);
                tWait[0].start;
                state := state2;
                ...........
        }
        [] tWait.timeout -> index i
        {
                // the call of user i has failed
        }
        ...........
}
```

However, the proposed change in the syntax of TTCN-3 requires changes in the compiler and in the execution runtime. To avoid this, another solution based on the Test Adapter (TA) is possible. The timers are no longer started from TTCN-3 from the TA which controls all **send** operations. The TA starts a timer by using the execution environment programming elements, e.g., timer in Java. The timeout is a pure event which is enqueued in the port used for communication with the SUT. The event contains also the information needed to process the call, i.e., the user to which the timeout event belongs to. The concept is presented in Figure 4.1.

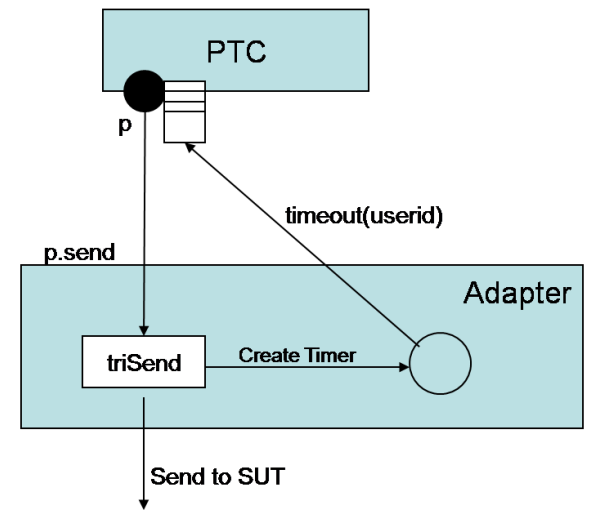

Figure 4.1: Timers Management in the Test Adaptor

4.5.7 Verdict Setting

The handling of verdicts in performance tests is different from the traditional verdict handling procedure in functional testing. In functional testing, the built-in concept of verdict is used to set a verdict when an action influences significantly the execution of the test, for example, if the SUT gives the correct answer the verdict is set to **pass** else if the response timer expires the verdict is set to **inconc** or to **fail**.

Performance tests have also to maintain a verdict which should be delivered to the tester at the end of a test case execution. The verdict of a performance test has rather a statistical meaning than only a functional one; still, the verdict should be a sum of all verdicts reported by test components. In our approach, the verdict is set by counting the rate of fails during one execution. If during the test, more than a threshold percentage of clients behaved correctly, the test is considered as passed. The percentage of correct behaviours in a tests must be configured by the test engineer himself and must be adapted separately to each SUT and test specification.

The collection of statistical information like fails, timeouts, successful transactions can be implemented by using counter variables on each component. These numbers can be communicated at the end of the test to a central entity, i.e., MTC which computes the final results of the test. If the test needs to control the load based on the values of these variables, than central entity must be periodically updated.

A simple convention for establishing the verdict is the following:

- *pass* - the error statistics are above the established threshold. The test sets the verdict to pass since it fulfils the performance test condition.
- *inconc* - the test did not execute successfully a preamble operation, e.g., cannot connect to the SUT.
- *fail* - the error statistics are below the threshold.

However, following this convention, it requires that the tester sets the verdict only at the end of execution after collecting and analysing all calls. Another option would be to modify the semantic of TTCN-3 verdict *to compute the verdict on the fly*. Each component modifies the verdict for each terminated scenario. At each modification the global verdict is recomputed.

4.6 Platform Architecture for TTCN-3

The design of next generation TSs considers obligatory to implement modular, software defined test architectures based on widely adopted industry standards [Ins06], [GHR+03]. One of the most important contributions to test execution field is the standardised TS architecture from ETSI, which presents a guideline for how test systems can be designed. The architecture has been defined for the TTCN-3 test specification language but it can be also used as a generic test execution architecture.

The general structure of a TTCN-3 TS is depicted in Figure 4.2. A TTCN-3 TS is build up from a set of interacting entities which manage the test execution. The specification consists of two documents: TRI [ETS07b] and TCI [ETS07c]. The TCI provides a standardised adaptation for

management, test component handling and encoding/decoding of a TS to a particular test platform, i.e., it specifies the interfaces between Test Management (TM), Component Handling (CH), Coder/Decoder (CD) and Test Executable (TE) entities. The TRI specifies the interfaces between the TE and TA entities, respectively. The two interfaces provide an unified model to realise the TTCN-3 based systems [SVG02], [SVG03].

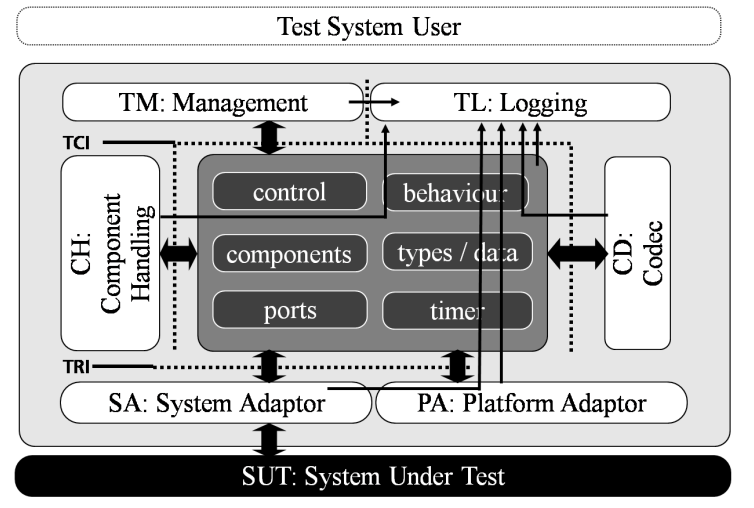

Figure 4.2: TTCN-3 Architecture of a Test System

The main components of the TS are:

- *Test Executable (TE)* - is the executable code produced by a compiler. The compiler transforms the TTCN-3 elements into platform specific elements, e.g., threads, classes. This component manages different test elements such as **control, behaviour, component, type, value** and **queues**, which are the basic constructors for the executable code.

- *Component Handling (CH)* - handles the communication between test components. The CH API contains operations to **create, start, stop** parallel test components, to establish the connection between test components, i.e., **connect**, to handle the communication operations, i.e., **send, receive, call and reply**, and to manage the verdicts, i.e., **setverdict**. The information about the created components and their physical locations is stored in a repository within the execution environment.

- *Test Management (TM)* - defines operations to manage tests, to provide and set execution parameters and external constants. The test logging is also collected by this component. The TM functionality presented to the user provides means to **start/stop** a test case or a whole test campaign and to monitor the test execution.

- *Test Logging (TL)* - performs test event logging and presentation to the TS user. It provides the logging of information about the test execution such as which test components have been created, started and terminated, which data is sent to the SUT, received from the SUT and matched to TTCN-3 templates, which timers have been started, stopped or timed out, etc.

- *Coder/Decoder (CD)* - in a TTCN-3 TS the data is encapsulated in a data model instance. A message instance is a tree of objects where the leaf nodes contain values of basic types,

e.g., **charstring, integer**. The non-leaf nodes represent structured data, e.g., **record**. This representation helps the test logic to better process the test data. Therefore, for sending data to SUT, a coder is needed to serialise the data into SUT understandable messages. On the receive way, a decoder makes the inverse process to transform the SUT message into a tree-like model instance.

- *System Adapter (SA)* - realises the communication with the SUT. The entity provides the communication operations **send, receive, call, getcall, reply**.
- *Platform Adapter (PA)* - implements the timers and the external functions. Timers are platform specific elements and have to be implemented outside the TS. The Platform Adapter (PA) provides operations in order to handle timers: **create, start, stop**. External functions, whose signature is specified in the TTCN-3 specification, are implemented also in the PA.

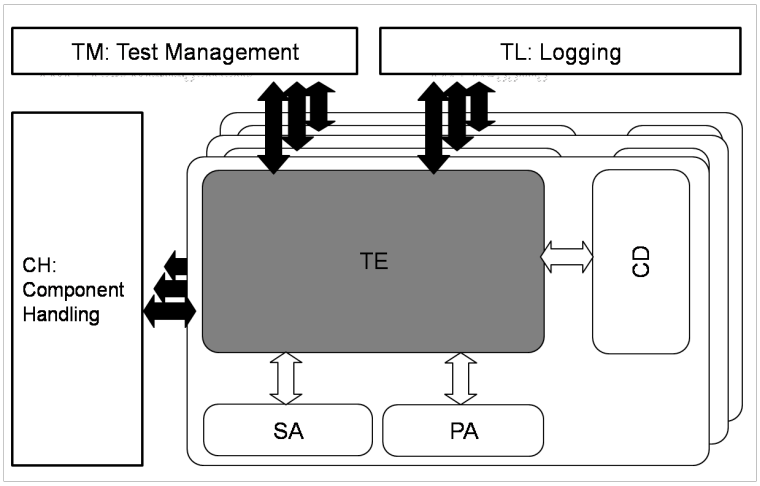

Figure 4.3: Architecture of a Test System for Distributed Execution

The TS can be distributed over many test nodes and different test behaviours can be executed simultaneously. Figure 4.3 shows the distributed perspective of the TS architecture. As it is conceived within the standard, on each host, an instance of the TS is created. The TE must be distributed on each host with its own CD, System Adapter (SA) and PA. The CH supplies the communication between the test components which run on different hosts.

4.6.1 Implementation Architecture of the TTCN-3 Execution Environment

The TRI and TCI interfaces provide a unified model to realise the TTCN-3 based systems [SVG02], [SVG03]. Despite their generality, from a technical point of view, the ETSI standard does not cover further aspects, required by a concrete test architecture [Alb03], such as test deployment, test distribution or test container service. Therefore, an extension of these concepts is needed. This section extends the concepts and the implementation architecture of a distributed TTCN-3 TS.

The architecture presented in Figure 4.4 lays out the entities of the extended test execution platform. The entities defined in the ETSI architecture are also distinguished in this architecture as

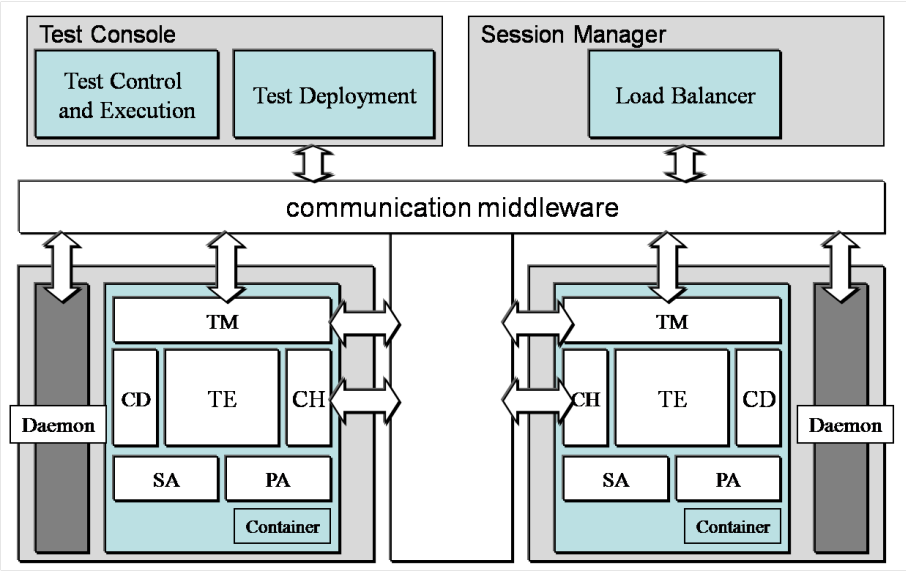

Figure 4.4: Implementation Architecture of the TTCN-3 Execution Environment

part of the special execution environment called *test container*. The architecture presents two test containers, as the test distribution is foreseen. Several test containers can be used in the same test session coordinated by a global *Session Manager (SM)* entity.

The *Test Console (TC)* is the point of the platform which offers the control functionality to the tester. The *test daemons* are standalone processes installed on any hosts to manage the test deployment activities. The test containers intercede between TC and TTCN-3 PTCs, transparently providing services to both, including transaction support and resource pooling. From the test view, the containers constitute the target operational environment and comply with the TCI standard for TTCN-3 test execution environment. Within the container, the specific the TS entities are recognised.

The main goal is to generalise the relationship between components and their execution environments. The interaction between two components or a component and a container is a matter of transaction. The transaction context provides runtime access to machine resources and underlying services. The transactional support must be isolated from the technical complexity required in development of component-based applications. These operations and many others are the target activities of the test containers.

In the following sections, four main roles are highlighted: *test management*, *test execution*, *test distribution* and *test logging*. For each role the contribution of the entities and their collaboration to fulfil certain activities is discussed.

4.6.2 Test Management

The *TM* includes all operations to deploy test resources, execute tests, collect log files and remove temporary files. In the proposed architecture, the test management is fulfilled by four components: TC, test daemon, test container (especially the TM-TE relation) and Session Manager (SM). These components are highlighted in Figure 4.5.

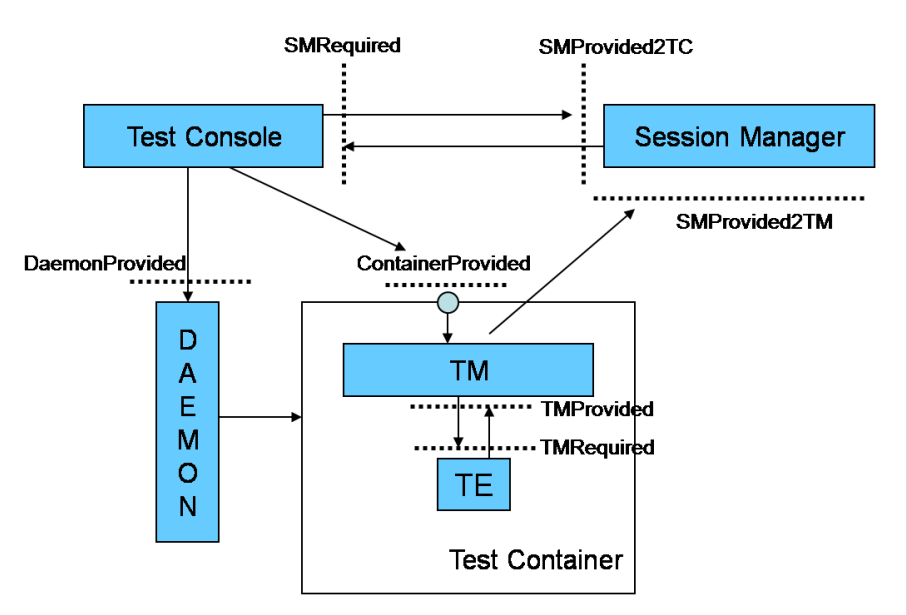

Figure 4.5: Test Management

4.6.2.1 Test Console-Session Manager Interaction

The *TC* is the entity that handles the management operations to create test sessions, deploy resources into containers and launch the test cases execution. It is the entity which provides a user interface and offers commands and actions to control the test environment.

A TC can be presented to the user in different forms:

- *command line with scripting support* - this is an old-style type of user interfacing but for performance testing it is perhaps the most efficient. The command interpreters do not consume as much resources as GUIs. The command line combined with scripting offers a very flexible technology. Different to GUI approaches, this concept allows the tester design combinations of commands which are not typically possible with GUI based tools. In general, GUIs offer a limited number of command sets which are mostly based on usage patterns.
- *GUI* - many tools use GUI frameworks such as swing or eclipse to define more user friendly user interfaces. The advantage against scripting is obviously the learning curve: it is much easier to control and avoid mistakes when using a GUI.
- *Web GUI* - a simpler form of GUI is Web GUI. This is the middle way between command line and GUI, which offers a Web based GUI to control the test environment. Web GUIs are not so evolved as normal GUIs but provide anyway a more user friendly interface than a command line console.

Independently on how the TC offers the control actions, GUI or command line, three categories of actions are identified.

Test sessions management. The TC activities are performed in the context of a test session. A test environment, especially a distributed one, offers the test resources to more than one test engineers.

Therefore, the test session concept is needed to distinguish between the test activities of different test engineers. Additionally, a tester may schedule different test sessions. The test environment has to offer an adequate interface to create and manage test sessions. The sessions' scheduler should take care that a test session has enough resources available. Therefore, one approach is to execute the sessions sequentially. The typical operations related to session management are:

- *create/kill* - at session creation, the TC makes the initial reservation of resources, e.g., deploy test files on all selected test nodes, but does not start the actual test execution. This is a separate step. After creation, the session is configured with the test parameters, test distribution strategy, etc.
- *status check* - a session goes through different states: initialised, running, terminated or error. The TC should offer the possibility to investigate the status of a session. Multiple views are possible: view of a certain session (by ID), view of all sessions of one user, view all sessions.
- *save/load* - a session can be saved and reloaded. This concept is very helpful to reuse an existent configuration. The basic idea is to reuse a configuration, i.e., same test nodes and test behaviour, for different test parameters.

The Session Manager (SM) is the place where all information about sessions is stored. It includes data such as who created the sessions, the current status of the session. It provides an API to the TC which may create, start/stop, kill each session.

Session control. Once a session has been created, it can be used to execute tests. For this purpose, further operations are required:

- *configure test component distribution rules* - in a distributed environment, the test engineer should be able to define rules for how the test components should be distributed. Additionally, the test engineer should be able to select load balancing algorithms.
- *configure test parameters* - to prepare the test for execution, test parameters have to be provided.
- *start/stop test* - after all preliminary configuration operations are finished, the test can be started. During the execution the tester may stop the test anytime.
- *on-line tracking* - the log statements will be logged during the execution. However, the tester may want to see these statements on-line while the execution progresses.

To allow the TC to administrate the test sessions, the `SMProvided2TC` interface has been introduced. The interface defines operations for all tasks presented above. The interface can be addressed by multiple TCs since many users can access the environment at the same time.

Another functionality the Session Manager (SM) may provide, is a call back interface to be able to call the TC when certain events happen. For instance, a callback event is the termination of a test. From the test console, the tester may start a wait-for-termination command which will actually wait until the Session Manager (SM) announces the termination. Since multiple test consoles may be connected at the same time to the platform, e.g., more than one tester, the callback implementation has to keep track of all consoles which are interested on certain events. The call back functionality is defined on the Test Console side within the SMRequired interface.

Test environment management. Test environment management implies the actions to administrate the standalone processes, i.e., the test daemons and the Session Manager (SM). This functionality has to be supported also by the test console. Typical commands are: `update`, e.g., to update the test daemon software, `version` (to check the current version), `reboot` (to restart the test daemons or the Session Manager (SM)), `ping` (to check the availability of a test daemon on a given test node). These operations are defined in `DaemonProvided` (for the test daemon) and `SMProvided2TC` interfaces.

The *time* is an important notion for distributed TS. A major problem is that there is no concept of the usage of a common clock. As a distributed system contains distributed test nodes where each node has its own notion of time, i.e., an own internal clock. The problem that occurs is that the clocks very often get un-synchronised altering the test execution. To coupe with this problem a simple technical solution is the Global Positioning System (GPS). Each test node receives the time from a GPS receiver, with a high accuracy, and substitutes the existing time operations. This way the time will be always the same for all test nodes.

Script Example. Listing 4.11 offers an example of a script to create, configure, execute and terminate a test session. These operations are explained in detail. The `ses_id` and `verdict` variables are used to save the id of the session and the verdict of the test. The `make` command creates a session using the configuration parameters in the `containersConfig.ccf` file and stores the id of the session into the `ses_id` variable. The session id is used in the `config` command to configure the distribution rules, in the `load` command to load the test parameters and in the `log` command to set the logging to "on". Setting the log to "on" means that the tester can see the log events displayed along the test execution. Setting it to "off" means that the events can be visualised only after the termination of the test. The `starttc` starts the test with the name `MyTest`. The `wait` command blocks the script execution until the session terminates. At termination, the verdict is stored into the `verdict` variable. The `log` command asks the execution environment to collect all log events and put them together into the `exec.log` file. At the end, the session is terminated and resources released, e.g., temporary files are removed, with the command `kill`. The `exit` command leaves the script and displays the final verdict.

Listing 4.11: Script Example

```
echo on

var $ses_id
var $verdict

# create a new session
make -ccf containersConfig.ccf -v $ses_id

# configure the new session (configure the deployment rules)
config -s $ses_id -tcdl distributionConfig.tcdl

# load test parameters
load -s $ses_id -mlf testParameters.mlf

log -s $ses_id on

# starts the session
starttc -s $ses_id MyTest

# wait until the execution terminates
wait -s $ses_id -v $verdict
```

```
log -s $ses_id -o exec.log

kill -s $ses_id

echo termination -verdict $verdict

exit -v $verdict
```

4.6.2.2 Test Console-Test Daemon Interaction

Test daemons are standalone processes installed on the test nodes (one test daemon per node) with the role to manage the test containers. A test session involves at least one test daemon during the execution. The test daemons are addressed by the TC during the test session creation. At session creation, each test daemon makes the initial resource reservation, e.g., creates the folders where the test resources will be deployed, creates the log files. After the creation of the container, the test daemon does not need any further interaction with the test daemon excepting the case when the TC sends the request to kill a session. During the test session administration, e.g., initialisation, parameterisation, start/stop test, the TC talks directly to the container. This functionality is provided by the test daemon through the DaemonProvided interface.

4.6.2.3 Test Console-Test Container Interaction

The test containers are the hosts of TEs; they manage the creation, configuration, communication and the removal of the parallel test components. Moreover, the containers are the target test execution environment and comply with the TCI standard for TTCN-3 test execution environment. The container includes the specific TS entities: TM, CD, TE, CH, SA and PA. For more information on the API and interactions between these entities, the reader is referred to [SVG02], [SVG03]. The container subsystems are functionally bound together by the TCI interfaces and communicate with each other via the communication middle-ware.

The TC addresses the test containers via `ContainerProvided` interface. During the test setup, the console communicates with all test containers created for a session in order to keep a consistent configuration and parameterisation. One of the test containers is marked as "special" container. The special test container is the one which creates the MTC component where the testcase will be started. When the testcase is started, the TC asks the "special" container for the TM interface which is then used to start the testcase within the local TE.

4.6.2.4 Test Management-Test Executable Interaction

The TM entity is responsible for the control of TE. It provides the necessary functionality for the TC so as to control the test execution. The functionality includes: start/stop of test cases, collection of final verdict and propagation of module parameters to the TE. In a test session, the TC addresses each TM on each test node in order to maintain a consistent configuration.

The interaction TM-TE consists of two sets of actions grouped into `TciTMProvided`, which is the functionality provided by TM to TE, and `TciTMRequired`, which is the functionality offered by TE but required by TM. Figure 4.6 shows some of the possible interactions between TM and TE. The two actions, `tciStartTestCase` and `tciStopTestCase` belong to the `TciTMRequired` while the `tciGetModulePar` belongs to `TciTMProvided`.

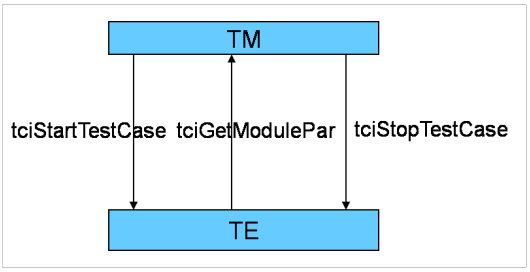

Figure 4.6: Test Management-Test Executable Interaction

The TM may start or stop the test cases and the control part. This functionality is implemented in TE. In response, the TE may notify the TM when a test case or the control started or terminated. The test parameterisation is also a task of the TM which includes the `tciStartTestCase` operation which has a `TciParamterListType` as argument.

The values of the parameters of a module are transferred to TE also via TM interface. TE may ask TM about the value of a parameter by using the `tciGetModuleParameter` function.

TM can obtain information about the content of a module directly from TE. With the `tciGetTestCases` and the `tciGetModuleParameters` functions, the TM may acquire the list of test cases and module parameters.

TE-TM relation has an important role also for logging and error handling. The TEs forward the logging events to the TM using **tciLog** function and it informs the TM about errors occurred during the runtime by using **tciError** function.

4.6.2.5 Test Management-SessionManager Interaction

The `SMProvided2TM` interface defines the operations which a TM can invoke on the SM. The operations are related to logging of `logStatements` and errors which are dispatched by the SM to all listening test consoles. Additionally, in case that errors are reported by any of the TM components, the SM updates the status of the test sessions to “error”. When a testcase starts or finishes its execution, the TM where the testcase runs, informs the SM about this event as well.

4.6.3 Test Execution

The part of the test platform which executes the TTCN-3 statements is the TE entity. Technically, TE contains the same test logic as the TTCN-3 code, but into a concrete executable form, i.e., executable code, produced by a compiler. The statements are usually expanded into bigger pieces of code, e.g., Java classes, which call the underlying platform functions to execute the test logic.

Some operations cannot be executed directly by the TE. Though TE is a concrete executable code, some function calls are only pointers to concrete functions which deal with the aspects that cannot be extracted from information given in the abstract test. For example, the mapping of the send statement does not know how to send the data to SUT and it needs to call a function provided by the SA. This non-TE functionality can be decomposed into:

- communication with the SUT provided by SA

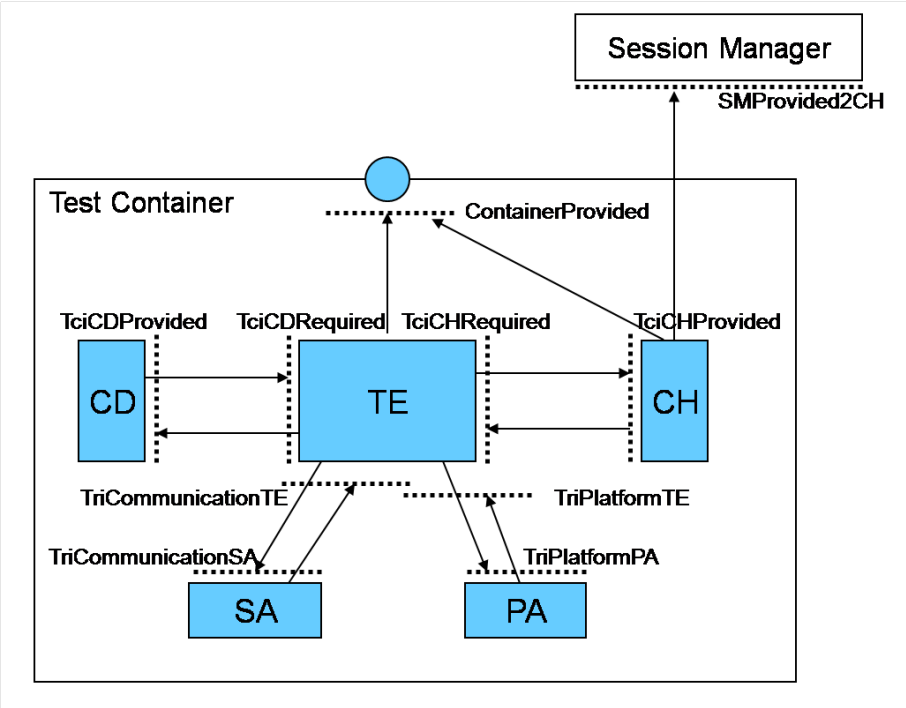

Figure 4.7: Test Execution

- external functions provided by PA.
- timer functions provided by PA
- component handling provided by CH and SM
- data encoding provided by CD

A very simple and incomplete mapping from TTCN-3 to Java is given as example in Table 4.2.

The test can be distributed over many test nodes and different test behaviours can be executed simultaneously. On each host, an instance of the TS is created. The TE must be distributed on each host with its own CD, SA and PA. The CH supplies the communication between the test components created, that run on different hosts. The TM coordinates and manages the execution of the test.

Figure 4.7 highlights the entities which contribute to the test execution. The interaction between these entities is defined as a set of interfaces. The API, conceived within the interfaces, defines operations which the entities either provide or use. For this reason, there are two types of interfaces: provided and required. The provided interfaces contain operations which are offered by an entity to TE; the required interfaces contain operations that a non-TE entity expects from the TE entity.

Table 4.2: Example of Mapping Rules from TTCN-3 Abstract Elements to Java Language Elements

TTCN-3 Element	Mapping to Java
basic type	a class which encapsulates the mapping of the type to Java types. The class provides access methods based on `TciValue` interface to the internal data.
structured type	a class whose fields are references to other types. The references are accessed through `TciValue` interface methods.
component	a class which provides references to ports and local variables.
function	a thread whose run method contains the mapping of statements. The component on which the function runs is a local field used to access the components' internals.
send	a function which checks whether the sending port is mapped to an SUT port or to another PTC port. If the destination is the SUT the function uses the reference to the SA and calls the `triSend` If the destination is another PTC, the function calls `tciSendMessage` of CH.

4.6.3.1 The ContainerProvided Interface

The test container has an important role in the test execution offering repository functionality for keeping the references to components, ports and functions. Additionally, it provides also the location information of entities which reside on other test nodes, e.g., Session Manager (SM), the CH of another test container. The functionality of the container is defined within the `ContainerProvided` interface.

To understand the repository role of the test container, the PTC management functionality is used as an example. Along the testcase execution, a PTC is created, mapped to SUT or connected to other PTCs, started and stopped or terminated. Whenever a component is used, a reference to its object is needed. This reference is managed by the test container. At creation, the component receives an ID and an instance of its component type is created. The container repository stores the instance together with its ID. Within the test behaviour, the components are addressed by their references, e.g., **self**, **mtc**, or a variable of type component. At the start of the component, the behaviour running on the component receives the component reference but not the concrete instance. To access the internal data of the component, e.g., variables, ports, the container is asked to *resolve* the ID into a concrete reference which is then used to access the internal data.

In a similar way, the container provides the functionality of a repository for ports and behaviours. For the port references, the container resolve their components to which they belong. The behaviours are searched in the repository when, for example, an error occurs and all threads have to be stopped.

The container keeps also all references to the entities TE talks to: SA, PA, CD and CH. These entities are used during the compilation process to provide the pointers to the interfaces they provide.

4.6.3.2 Test Configuration

A test configuration consists of all components created by a testcase and the connections between the ports of different components. The TTCN-3 operations to create a test configuration are: **create, map, unmap, connect** and **disconnect**. The compiler maps these operations to function pointers whose implementation is provided by CH. The TE itself cannot supply this functionality since it does not have the knowledge about where the components are located. A component can be created either locally or remotely, on another test node. To acquire this information, the CH is used as proxy through its `TciCHProvided` interface.

The decision where a component has to be created is taken by the SM which supplies the rules for PTCs distribution. For this purpose, the SM provides the `SMProvided2CH` interface. At component creation, the CH asks the SM for a the location of the new component. After the creation, the ID of the component contains also the location information, therefore, for other configuration operations, the CH can manage alone where the operation will be executed.

The configuration operations are called from test cases or functions. CH keeps track of the modifications in the TS configuration. It stores the references of the components, knows about the running behaviours and manages the component connections. CH can also be asked if a component is running or if it finished its execution. Whenever an execution error occurs inside of a TE, the `tciReset` operation is called in order to stop all the other components of all TEs.

In a distributed environment, the CH also helps in deciding if the requests are to be handled locally or remotely. If the request is executed locally, it is sent back to the local TE. TE provides the `TciCHRequired` interface so as to allow CH to address a TE. If the request is to be executed on a remote node, the CH sends the request to the remote TE.

4.6.3.3 Inter-Component Communication

The communication operations between components are also realised by CH. CH provides to TE a set of operations (in `TciCHProvided` interface) which are called whenever data is transmitted between components. The CH communication operations represent the mapping of the TTCN-3 communication operations, e.g., **send, receive, call, getcall**. When a component sends data to another component, the request is forwarded to CH. CH has the global view over the distributed configuration and knows the host of the receiving component as well. If the host is a remote one, the message is send to the remote TE, which offers the `TciCHRequired` interface. TTCN-3 supports two types of communication: asynchronous (message based) and synchronous (procedure based). Consequently, CH supports also the two types of data communication by providing methods like `tciSendConnected, tciEnqueueMsgConnected` for asynchronous communication, but also methods like `tciCallConnected, tciEnqueueCallConnected` for procedure based communication.

4.6.3.4 Test Executable-Coder and Decoder Interaction

The TTCN-3 values must be converted to SUT specific data, i.e., **bitstring**. This task is solved by CD, which provides the `tciEncode` and the `tciDecode` operations. These functions are called by the TE when it executes **send** and **receive** operations. The interaction between TE, CD and SA is illustrated in Figure 4.8.

The encoding operation is called by TE before the data is sent to the TA. The decoding operation is

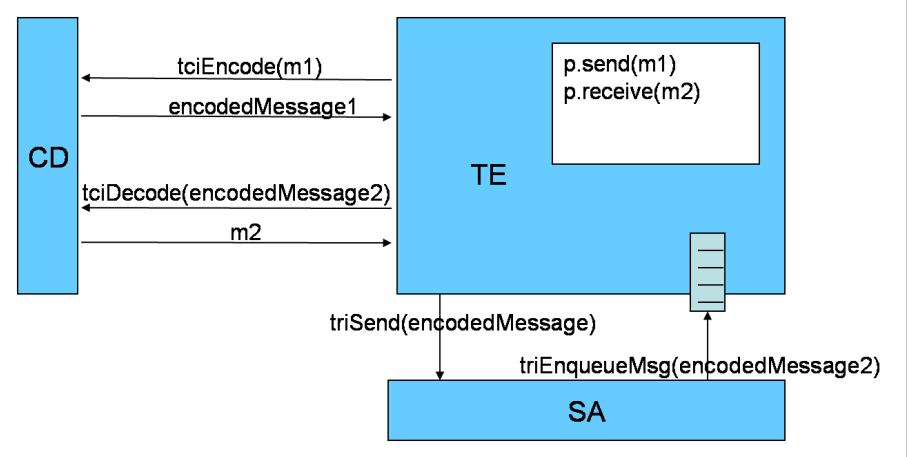

Figure 4.8: Coder and Decoder Functionality

used by TE when data is received from the TA. To be able to decode, CD requires some information from TE; this information is the *decoding hypothesis*. The behaviour running in TE knows the type of data that is to be received, so it asks CD to try to decode the received data according with a certain expected type. If the decoding fails, CD is called again with a new decoding hypothesis.

During the encoding, the TTCN-3 data is transformed into concrete data values as required by the SUT. The TE contains the data values which are generated at compilation in form of a tree of objects. In order to maintain a common data model, the TCI standard defines the *Abstract Data Type Model*, which describes the interfaces that handle the TTCN-3 data from TE. Each abstract data type has associated a set of operations which allows for accessing its internal data.

The *Abstract Data Type Model* contains two parts: the data type `Type`, which represents all TTCN-3 types, and data types that represent TTCN-3 values (instances of TTCN-3 types). All TTCN-3 types provide the same interface. One may obtain for a type the module where it was defined, its name or class and the type of encoding. Any type can be instantiated by using the `newInstance` method. The different types of values, which can appear in a TE, are presented in Figure 4.9.

All types of values inherit the `Value` type and provide the core operations: `getValueEncoding`, `getType` and `notPresent`. The `Value` types represent the mapping of the abstract TTCN-3 values to concrete ones. There are three categories of values: a) basic types, e.g., **integer, float, boolean**, b) string based values, e.g., **hexstring, octetstring, charstring**, and c) structured types, e.g., **union, record, verdict, etc.**. Besides the core operations extended from `TciValue` interface, all of them provide additional operations, specific to each type, to access their content.

4.6.3.5 Test Executable-System Adapter and Test Executable-Platform Adapter Interaction

The TRI interfaces define the communication of TE with the SA and with the PA. The PA is a gateway to host resources, i.e., timers, external functions. The SA defines a portable service API to connect to the SUT. The TA is a user provided part and implements the SA interface. TAs promote flexibility by enabling a variety of implementations of specific tasks.

This part describes two tasks: firstly, the way the TE sends data to SUT (`TriCommunicationSA`

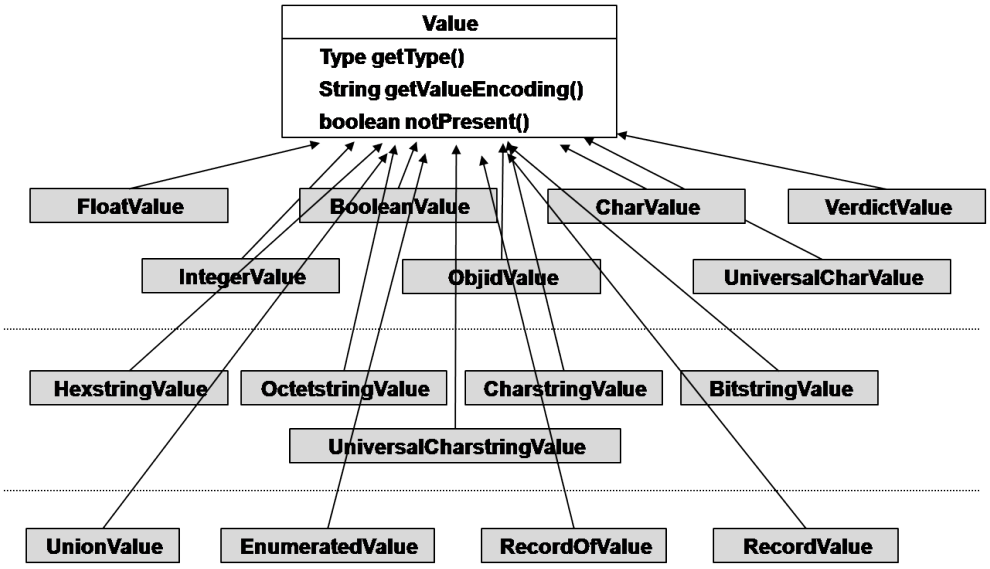

Figure 4.9: The TTCN-3 Value Types which can Appear in Test Executable

interface) or manipulates timers and external functions (`TriPlatformPA` interface), and secondly, the way the SA notifies the TE about the received test data (`TriCommunicationTE` interface) and PA notifies the TE about timers' timeouts (`TriPlatformTE`).

The TTCN-3 specifications use the operation **system** to reference the SUT. The component returned by the **system** operation is conceptually similar to the other PTCs and is usually called **system** component; it may contain ports which can be connected to other components. In contrast to normal components, the ports of the **system** component are connected with the ports of other components by using the **map** operation. As far as the test execution is concerned, the **system** is handled differently. The configuration operations **map, unmap** are not sent to CH, but to the TA.

SA implements all message and procedure based communication operations. The data received from SUT is transmitted to TE also through SA. The enqueue operations are implemented by TE, so the TA implementation shall be concerned with the task of using the enqueue operations to deliver the received data to TE.

4.6.4 Test Distribution

As far as the test distribution is concerned, TTCN-3 provides the necessary language elements for implementing distributed tests as well. This is, however, supported in a transparent fashion as far as the same test may run either locally or distributed. The distributed execution of a test enables the execution of parallel test components on different test nodes, sharing thus a bigger amount of computational resources.

The design of a distributed TTCN-3 test architecture rises a number of conceptual questions [Apo04]:

- *What does it imply the distribution of a component on a specific node?* - a component consists basically of data and ports. The component is like an object which contains a number of other objects. The ports can be seen also as objects which contain a list of available

connections (to other ports). During the test execution, a component is started with a test behaviour. A test behaviour is a thread which has a reference to the component object in order to access its internal data.

- *How can it be specified that a component has to be distributed on a specific node?* - the TTCN-3 language lacks of support to specify which components should be executed to a remote test node. This issue requires a further development of distributed testing concepts related to TTCN-3.
- *How can a load balancing algorithm be applied?* - a performance test may create a large number of components. Moreover, the test nodes may have different amounts of CPU and memory resources. Therefore the question arises: How can one balance the components in an intelligent manner such that the test nodes with more resources run more parallel components?
- *How can it be specified which hosts shall participate in a test session?* - a test component may be distributed anywhere on any test node. The question is how to specify which hosts should be used.

4.6.4.1 Container Configuration File

To give the possibility to describe the participating nodes and the sources that have to be deployed, a Container Configuration File (CCF) concept is proposed. Every node is identified by an IP address and a logical name. The logical name is used to simplify later needed referencing for a user. The sources that have to be specified include the TE created by the TTCN-3 compiler, the TA together with the appropriate CD, and all other libraries needed by the test suite.

Listing 4.12: Filter by component type

```
<container_list>
        <container>
        <IP>192.168.99.18</IP>
        <name>host1</name>
        <deploy name="../../lib/CommonBehaviours.jar" />
        <deploy name="../../lib/CommonTemplates.jar" />
        <deploy name="../../lib/CommonTypes.jar" />
        <deploy name="../../lib/UserStateMachine.jar" />
        <deploy name="../../lib/adaptor.jar" />
        </container>

        <container>
        <IP>192.168.99.19</IP>
        <name>host2</name>
        <deploy name="../../lib/CommonBehaviours.jar" />
        <deploy name="../../lib/CommonTemplates.jar" />
        <deploy name="../../lib/CommonTypes.jar" />
        <deploy name="../../lib/UserStateMachine.jar" />
        <deploy name="../../lib/adaptor.jar" />
        </container>
</container_list>
```

An example of a CCF if given in Listing 4.12. A CCF starts with the tag `container_list`. For every container, which has to be created, a `container` tag is added. In the example, two containers shall be created. Within a container tag the host, the container name and the needed files have

to be specified. The host where the container has to be created is specified within the `IP` tag. The IP address of the host of the creating container has to be written. An IP address has to be used only once in a CCF as only one container on a host can be used in the appropriate test session. In the given CCF example containers will be created on the hosts with the IP addresses `192.168.99.18` and `192.168.99.19`. The IP addresses where a container has to be created and the container's names have to be unique in the CCF file. The sources that have to be deployed to all containers, and required in order to execute the test, are specified within the deploy tags.

4.6.4.2 Component Distribution Language

A test distribution strategy defines how the components are distributed among test nodes and thus it plays a major role in the efficiency of a TS. Test distribution defines which components are to be distributed and where they should be deployed. The distribution of components can be defined as a mathematical function of different parameters which is applied at deployment time separately for each test component in order to assign it to a home location where it will be executed. In the function definition (8), D is the distribution function, p1, p2, ..., pn are the parameters which influence the distribution and h is the home where the test component should be distributed.

There are two types of parameters which are taken into consideration when distributing test components: external and internal.

(8) $h = D(p1, p2, \ldots, pn)$

The external parameters are application independent parameters whose values depend on the execution environment and are constant for all applications running on that environment. Examples of external parameters are: bandwidth, CPU and memory.

The internal parameters are related to the test component based application itself and are different for each test case. They refer to characteristics which can be obtained at the creation of the test component: the component type, the instance number, component name and the port types which belong to that component.

When multiple homes can be selected for a particular sub-set, a balancing algorithm can be applied, e.g., round-robin. This mechanism encapsulates the complexity of distribution avoiding complex distribution rules. In fact, the goal of distribution is to rather help the tester distribute the components in an efficient way than to provide a language to implement any distribution rule.

Based on the aspects described above, a simple language for defining the distributions of test components has been defined. To help understanding the concepts related to test component distribution, some examples written in this language are presented here. The distribution specification is the process of assembling test components to hosts. The assembling process groups all components to be deployed, in a big set while the assembling rules shall define sub-sets of components with a common property, i.e., all components of the same type. A (sub-)set defines a selector of components and the homes where the selected components are placed. The filtering criteria of the selector handle component types, component instance numbers or even component names. The home are the possible locations where the test components may be distributed. Also for the homes further constraints for distribution can be defined, e.g., max number of components.

Figure 4.10 explains the principle of applying the assembly rules. A distribution strategy consists of a number of rules which are interpreted top-down. Each rule follows the flow presented in the

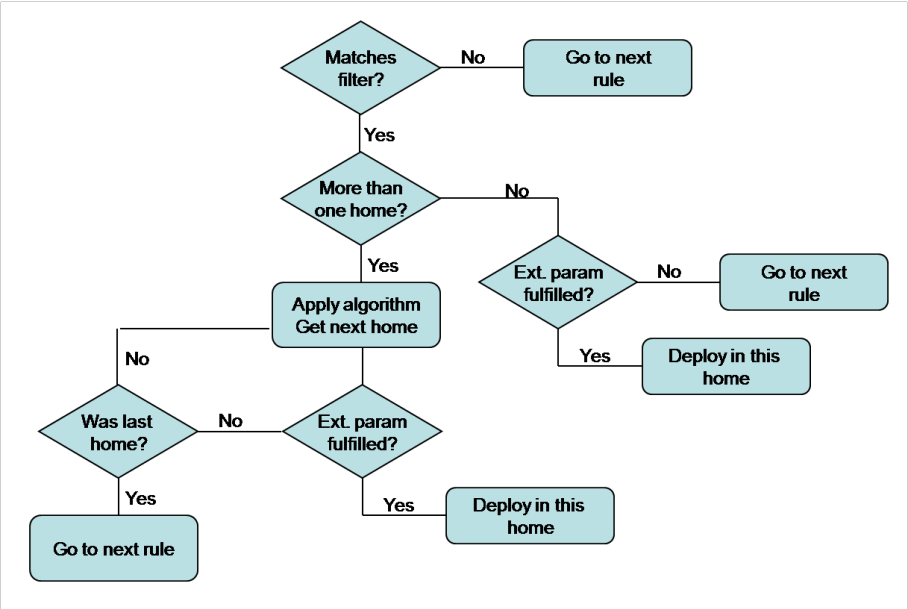

Figure 4.10: Home Finding Flow

figure. As a general rule, the internal parameters impose filtering rules and define which components to be selected while the external parameter validate whether a selected home can sustain a new component or not. The external parameters are always checked after the home has been identified. The flow starts with checking if the filtering rules match the component. If not, then the next assembly rule is taken. On successful match, it is checked if more than one home is specified in that assembly rule. If false, then it is checked if the external parameters for that home are satisfied. If not, then the next rule is taken; otherwise, the component is deployed in that home. If more than one home is specified, then a balancing algorithm must be specified too. This algorithm specifies in which order to choose the available homes. A simple algorithm is the round-robin algorithm which says that the homes are selected sequentially. If the external parameters of a home are not satisfied, then the algorithm is applied again for the rest of the homes until one home satisfies its external parameters. In the case no home can be used, then the next assembly rule is selected.

Filtering by component type. Listing 4.13 is an example of a component assembly rule based on filtering the components on their types. The tag with the name “special" indicates the host where the MTC component is deployed. The selector defines a filter to select all components of type `ptcType`. The selected components can be deployed either on `container1` or on `container2`. One can define deployment constraints for each container, for example, do not allow the deployment of more than 100 components on `container2`. In a similar way, other constraints related to memory usage, CPU load, number of components, etc., can be defined.

Listing 4.13: Filter by component type

```
<component_assembly>
    <description>Example to use TCDL language</description>
    <special container="container1"/>
    <set>
        <component_selectors>
            <componenttype>ptcType</componenttype>
        </component_selectors>
```

```
        <homes distribution="round-robin">
            <container id="container1">
                <max_components>10</max_components>
            </container>
            <container id="container2"/>
                <max_components>100</max_components>
            </container>
        </homes>
    </set>
</component_assembly>
```

Filtering by instance number. Listing 4.14 shows a set which deploys the instances 1, 2 and 5 of type `ptcType4` on the `container2`. The number of the instances are given as parameters in the filter rule.

Listing 4.14: Filter of instance number

```
<set>
        <component_selectors>
            <instance type="single">
                <componenttype>ptcType4 </componenttype>
                <number>1</number>
                <number>2</number>
                <number>5</number>
            </instance>
        </component_selectors>
        <homes distribution="">
            <container id="container2"/>
        </homes>
</set>
```

Filtering by component name. Listing 4.15 shows a set which deploys the component with the name `ptc1` on the `container2`. The number of the instances are given as parameters in the filter rule.

Listing 4.15: Filter of component name

```
<set>
        <component_selectors>
            <componentname>ptc1 </componentname>
        </component_selectors>
        <homes distribution="">
            <container id="container2"/>
        </homes>
</set>
```

Distribution by algorithm. Usually, the definition of constraints is a difficult task; for complex setups it may be very difficult to describe an efficient distribution. Therefore, the task of identifying hardware options and constraints should be realised by the test execution environment itself. It should provide services, which implement distribution algorithms that are designed to be efficient for a certain type of problems. The task of the user remains to select the algorithm which solves the problem best [Tol05].

Listing 4.16 provides an example of applying the `mem_prediction algorithm` [Tol05] for all components of type `ptcType`. This algorithm encapsulates the whole complexity of finding the memory proportions among all test nodes. It also does not require further constraints on the homes - it just provides the whole functionality by itself.

Listing 4.16: Distribution by algorithm

```
<set>
        <component_selectors>
            <componenttype>ptcType</componenttype>
        </component_selectors>
        <homes distribution="mem_prediction">
            <container id="container1"/>
            <container id="container2"/>
            <container id="container3"/>
        </homes>
</set>
```

The collector rule. The components which are not accepted by any set selector are deployed in a default home. This home is defined by collector tag.

Listing 4.17: Collector Home

```
<collector>
     <container id="container1"/>
</collector>
```

4.6.4.3 Distributed Component Handling

The assembly rules are implemented within the SM. Each session has assigned a set of assembly rules which are applied at each PTC creation.

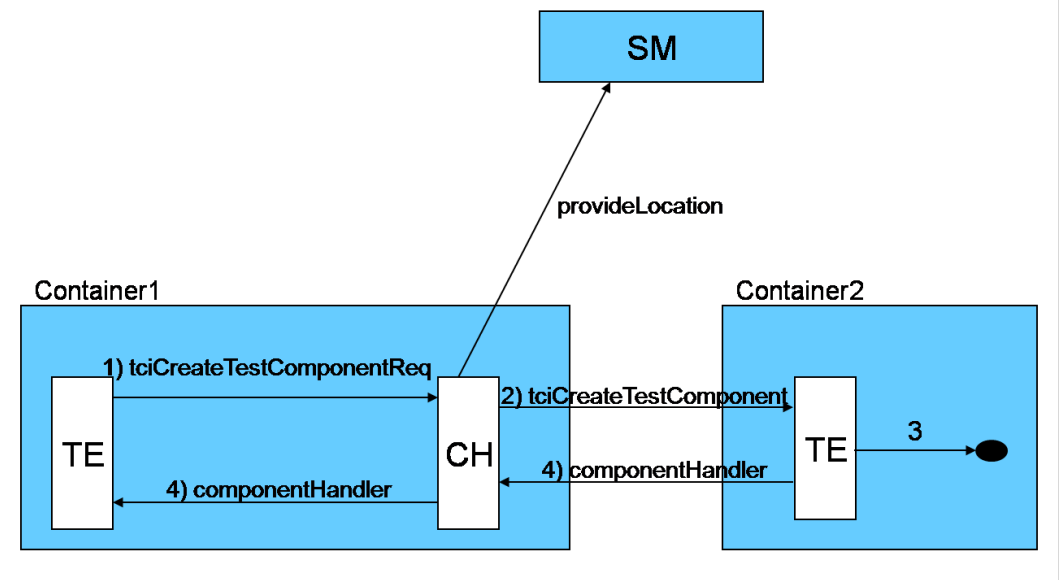

Figure 4.11: Component Creation

Figure 4.11 illustrates the mechanism of component creation. The scenario presented in the figure involves two test containers. The TE of `container1` intends to create a component on the second container. To realise that, the TE of `container1` invokes the `tciCreateTestComponentReq` on the CH of the same container. Next, the CH asks the SM for a location for the new component. Based on the SM's decision, the request for the creation of a component will be either transmitted

to the local TE or to a remote TE. In the scenario presented in the figure, the location is a remote container; therefore, the `tciCreateTestComponent` operation is invoked on the remote TE. The remote TE will create the TTCN-3 component and will provide back a handler to the requesting TE of `container1`. The local TE can then operate on the remote test component by using the obtained handler.

4.6.5 Test Logging

Log events are understood as single logical information units generated during the test execution. Log traces are physical representation of the logs, mentioned above. In relation to the test execution their order represents the whole test against considered System Under Test. Log traces are given to the user or test operator in textual format.

Analysis of such a textual log traces is done after test execution and implies manual or automatic review of previously generated text. Manual analysis requires a lot of time and effort as people involved have to understand every single detail and also recognise possible doubts, misunderstandings or errors. It is clear that in cases of huge test campaigns big effort is needed to analyse log traces. Automatic form, on the other hand, requires creation of appropriate tool, which could be reasonable in some cases.

The realisation of all mentioned issues requires a Test Execution Logging Interface. The task of the logging interface is to define a generic API provided by the test execution tool. The Logger is the client of this interface and handles the events produced during test execution.

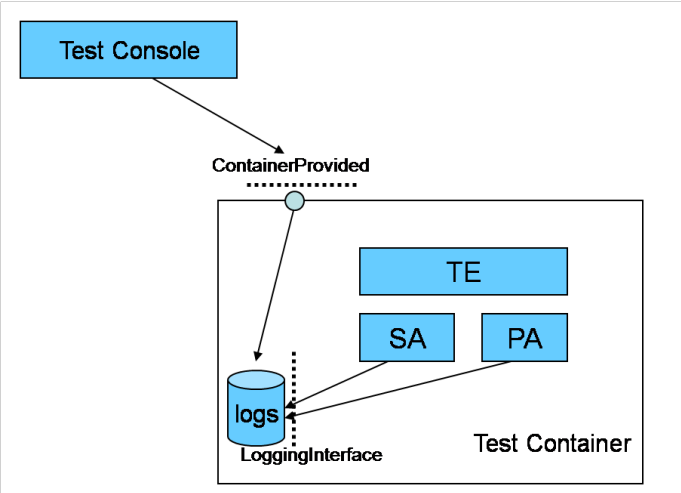

Figure 4.12: Entities Involved in Test Logging

Logging interface is a crucial part enabling observation of test log traces. It is a bridge between Test Execution Tool and Logger. It therefore provides all possible log events. Besides creation, start and termination of test components and ports several other events represent the operations done during the test case. The most important are the execution (start/stop) of test cases, the timers within statements: start, stop, timeout.

However, the afore-mentioned examples cover rather the test execution than the test logic itself. Especially, for performance testing, the information related to users transactions is far more im-

Listing 4.18: Log Event Format

```
[timestamp], [test node id], [component id],
[user id], [call id], [event type], [event content]
```

portant. Figure 4.12 presents the entities involved in the logging. Though other entities of the container, e.g., CH, TM, CD, may produce logs, the log events they produce are not relevant for performance testing. The information related to user states is captured by SA and PA entities. Therefore, the events to be stored in the log repository are: `triSend, triEnqueueMsg` and `triTimeout`.

The general form of an event is illustrated in Listing 4.18. Each event has a time-stamp. It also needs to store information about who produced it: `test node id` and `component id`. The rest of the information is related to user and its state: `user id`, `call id` and `event type`. It is also possible to store more information about the event, e.g., if a second user is involved in the call, then it can be also logged.

At the end of execution the TC uses the `ContainerProvided` interface to ask each container on each test node to provide the log traces. These traces are then sorted on timestamps so that the trace contains the events in there temporal order.

4.7 Summary

This chapter approaches the test harness topic in the performance testing context. It starts with the requirements related to test harness for the characteristics specific to performance testing (including test distribution). The TTCN-3 language has been selected to allow a more concrete illustration of the presented concepts. The arguments for the selection of this technology are also given. The language is then used to present how various performance testing concepts introduced in Chapter 3 such as event handling, traffic set, data repositories, etc., can be designed. Then the chapter converges toward execution platform concepts. An extension of the ETSI standardised architecture for test execution is discussed. Various aspects such as test management, test execution or distribution are discussed in detail.

Chapter 5

Case Study: IMS Performance Benchmarking

Ethical axioms are found and tested not very differently
from the axioms of science. Truth is what stands the test of experience.
– Albert Einstein

The performance testing methodology has been evaluated throughout a case study on IMS benchmarking. IMS [3GP06] is a standardised architecture designed to support a rich set of services and make them available to end users. The multitude of offered services and the complex call flows of the interactions between users and IMS network, make IMS an excellent candidate to apply the methodology. Therefore, the goal of the case study is to investigate how an IMS system will behave when the number of calls per second and the number of active users increase for a realistic selection of test scenarios.

The existing models inherited for legacy telephony, e.g., Erlang tables, 3-minute average holding time, 1 Busy Hour Call (BHC) per subscriber) [Ack04] are insufficient for IMS service model deployments. The service providers need IP based models more similar to those used for data networks and applications servers. This can be achieved only if the test system creates adequate workloads which consist of ordinary calls invoking the services.

The purpose of performance testing in the IMS context is to provide information on the complete IMS network and on subsystems of such a network corresponding to discrete products that may be available from a supplier. Therefore, the test system defines the input stimulus to the system under test by providing precise definitions of transaction types and contents, statistical distributions for transaction types, arrival rates, and other relevant parameters. The tests specify how the traffic is to be provided to the SUT and define the measurements to be made from the SUT and how they are to be reported.

5.1 Benchmark Definition

The goal of project, where the case study derives from, is to create a benchmark for IMS which implies the realisation of a representative workload and a test procedure capable of producing comparable results. Therefore, some definitions related to benchmarking are presented first.

A *benchmark* [JE05] is a program or an application used to assess the performance of a target system (hardware or software). This is realised by simulating a particular type of workload on a component or system. In a benchmark the test scenarios test the SUT services, the utility performance, one can experiment with different methods of loading data, transaction rate characteristics as more users are added, and even test the regression when upgrading some parts of the system.

A benchmark is a type of performance test designed such that it can be used as a method of comparing the performance of various subsystems across different chip/system architectures [VW98],[DPS07]. From a business perspective, a benchmark is the ideal tool to compare systems and make adequate acquisition decisions [Men02b]. Nevertheless, a benchmark serves also as a performance engineering tool in order to identify performance issues among versions of a system.

A benchmark is not easy to realise and often the tools have to deal with high performance capabilities. The list of requirements includes:

- *benchmark procedure* - the method for defining the load and for measuring the performance, should be based on the number of virtual users. This can be measured either by the number of subscribers a specific configuration can support, or the minimal system cost of a configuration that can support a specific number of users.
- *hardware resource limitations* - the benchmark must measure the capacity of a system. This requires the test system to emulate the behaviour of a big number of users interacting with the SUT. Although the benchmark will define a traffic load corresponding to thousands of simulated users, the method of traffic generation should be defined in such a way that it can be implemented economically by a test system.
- *realistic workload* - the benchmark must be driven by a set of traffic scenarios and traffic-time profiles, i.e., the benchmark is executed for various load levels. The test system should generate realistic traffic and cover the majority of use cases of interest.

5.2 IP Multimedia Subsystem

The IP Multimedia Subsystem is the 3GPP standardised architecture and protocol specification, based on Session Initiation Protocol (SIP) [IET05] and IP [IET81] protocols, for deploying real-time IP multimedia services in mobile networks [3GP06]. ETSI TISPAN has extended the architecture in order to support the deployment of IP services in all communication networks, e.g., fixed, cable. IMS provides the basis of a multimedia service model for core voice services (a.k.a. VoIP) and for new services based on voice, but including both video, e.g., video conferencing, and data services, e.g., location.

For users, IMS-based services enable person-to-person and person-to-content communications in a variety of modes including voice, text, pictures and video, or any combination of these in a highly personalised and controlled way [Eri04]. At the same time, for service operators, the architecture simplifies and speeds up the service creation and provisioning process, while enabling legacy inter-working.

The main layers of the IMS architecture are the *access layer, control layer*, and *service layer.*

- *access layer* - comprises routers and switches, both for the backbone and the access network.

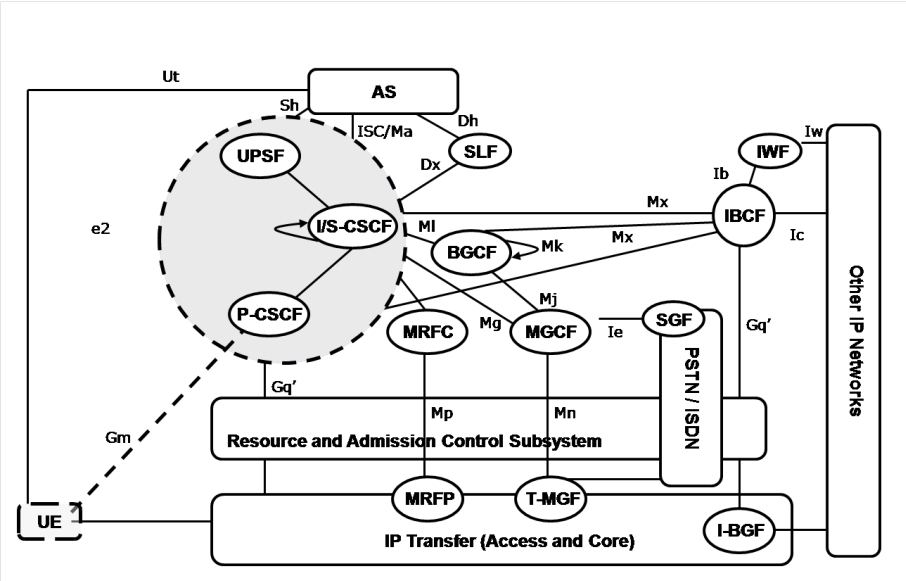

Figure 5.1: TISPAN IMS Architecture

- *control layer* - comprises network control servers for managing call or session set-up, modification and release. The most important of these is the Call Session Control Function (CSCF), also known as a SIP server. This layer also contains a full suite of support functions, such as provisioning, charging and and management. Inter-working with other operators networks and other types of networks is handled by border gateways.

- *service layer* - comprises application and content servers to execute value-added services for the user. Generic service enablers as defined in the IMS standard, such as presence and group list management, are implemented as services in a SIP Application Server (AS).

The IMS architecture is presented in Figure 5.1, where the call control signalling entities are highlighted. The key technology behind IMS is the SIP protocol, which is underlying many of the important interfaces between elements in an IMS-based network. The traffic between a UE and the Proxy-CSCF (P-CSCF) is carried by an IPSec [Dav02] tunnel corresponding to the UE. A P-CSCF is a SIP proxy, that is the first point of contact for the IMS terminal. It is assigned to an IMS terminal during registration, and it does not change for the duration of the registration. The Interrogating-CSCF (I-CSCF) queries the User Profile Server Function (UPSF) (or Home Subscriber Server (HSS) in 3GPP specifications) to retrieve the user location, and then routes the SIP request to its assigned Serving-CSCF (S-CSCF). An Subscriber Location Function (SLF) is needed to map user addresses when multiple UPSFs are used. For the communication with non-IMS networks, the call is routed to the Interconnection Border Control Function (IBCF) which is used as gateway to external networks. S-CSCF is the central node of the signalling plane; it decides to which ASs the SIP message will be forwarded to, in order to provide their services. The ASs host and execute services and are instantiated by S-CSCF. The Media Resource Function (MRF) provides media related functions, e.g., voice streaming. Each MRF is further divided into a Media Resource Function Controller (MRFC) for signalling and a Media Resource Function Processor (MRFP) for media related functions. The Breakout Gateway Control Function (BGCF) is a SIP server that connects IMS clients to clients in a circuit switched network. A Signalling

Gateway (SGW) interfaces with the signalling plane of the circuit switched network.

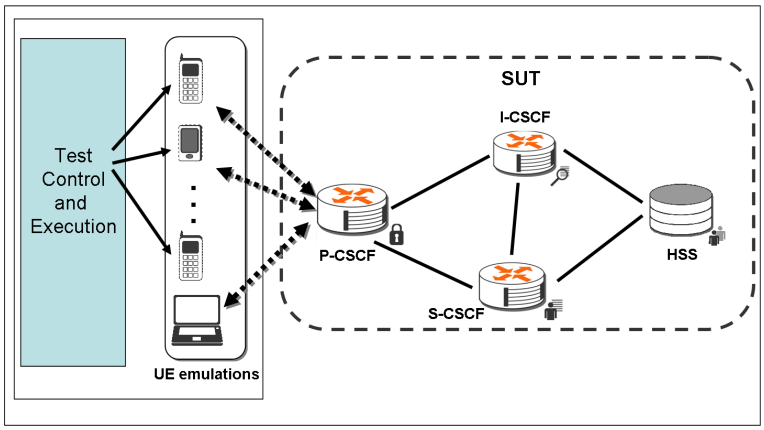

Figure 5.2: The IMS control layer

The main components of the IMS have been briefly introduced. However, the test system focuses on the performance evaluation of the core network components only (highlighted in the figure). The primary goal of the benchmark is to define a set of test cases that can be applied to an IMS SUT in order to evaluate its performance. The benchmark is defined for the control plane of an IMS network, which consists of the x-CSCF, HSS, and SLF components, the links over which they perform signalling, and the database transactions required to perform these functions. These elements are highlighted in Figure 5.2 which presents the test architecture in relation with the IMS control layer. A call is a relationship between one or more subscribers, signalled and managed through the IMS system. The intent of a subscriber when using IMS is to make calls, so the obvious way to obtain realistic behaviour in an IMS test is to model the behaviour of calls.

5.3 Session Initiation Protocol

The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, modifying, and terminating sessions with one or more participants. The SIP protocol has been developed by IETF and standardised as Request for Comments (RFC) 3261 [IET05] and it was accepted as a 3GPP signalling protocol of the IMS architecture. Within IMS network, SIP is used at the application and control layer. SIP is a relatively simple protocol. It is a text-based protocol allowing humans to read and debug SIP messages. A SIP message consists of three parts:

- *the start line* - this line determines whether the message is a *request* or a *reply*. The start line of a request contains the method name followed by the protocol name and version. The start line of a response starts with the protocol name and version.
- *the header fields* - the header fields consist of a list of headers where each field has a name followed by a colon and a value.
- *the body* - a blank line separates the header fields from the body. The message body is optional but when it is specified, the type of the body is indicated by the `MIME` value of the Content-Type header. The length is specified by the `Content-Length` header.

The original SIP specification defines six types of requests, i.e., `REGISTER, INVITE, ACK, BYE, CANCEL` and `OPTIONS`, and one type of response. The responses are differentiated by codes, as for example `100 Trying` (sent by proxies to indicate that the request was forwarded), `200 OK` (the general purpose OK response), `404 Not Found` (sent when the callee is not currently registered at the recipient of the request).

The SIP specification has been extended by various IETF documents, in order to increase its functionality. Some of the relevant SIP extension methods, used in IMS, are: `SUBSCRIBE` (to subscribe for an event of notification from a notifier, `NOTIFY` (to notify the subscriber of a new event; it is sent to all subscribers when the state has changed), `UPDATE` (update the session information before the completion of the initial `INVITE` transaction).

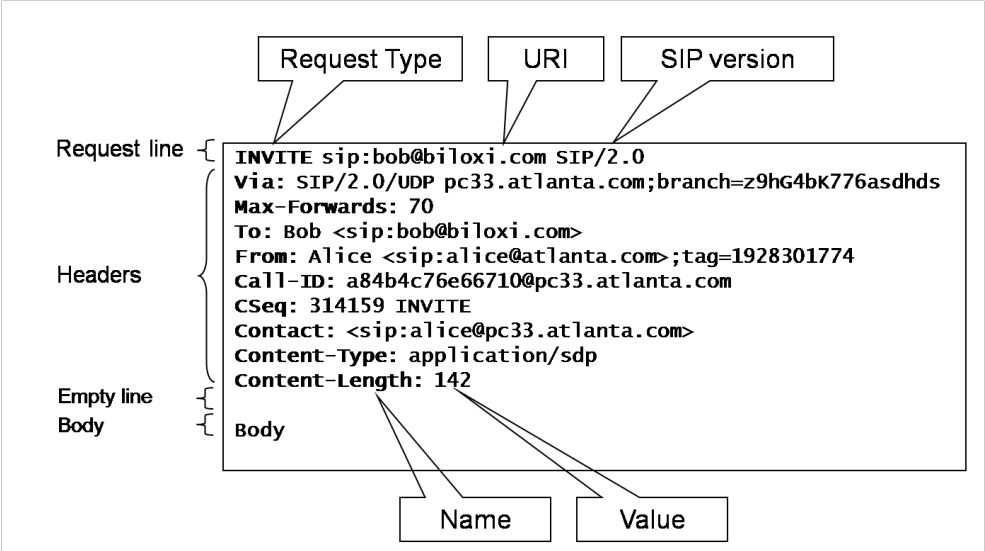

Figure 5.3: SIP Invite Request

The general structure of a SIP request is depicted in Figure 5.3. Each request has at least the following headers: `To, From, CSeq, CallID, MaxForwards, Via`. These fields supply the required information for services such as routing and addressing. The Request-URI field defines the callee where the request should go. The To field defines the receiver of the message. The From contains the identification of the sender. Call-ID is an unique identifier for a whole dialogue. The Call-ID is generated at the creation of each call; all messages which belong to a call have the same Call-ID. The CSeq serves for identification of a message within a call and defines the sequence of the messages. The Max-Forwards defines how often the message should be forwarded until it reaches the receiver. The Via field identifies the entities from the IMS network which receive the message.

5.4 Use-Cases, Scenarios and Scenario based Metrics

The benchmark traffic has been defined by selecting scenarios from the most common IMS use cases that are encountered the most in the real life deployments. For each use case a number of scenarios have been defined and for each scenario, its design objectives have been identified. However, further use cases can be defined in a similar way [Din06].

5.4.1 Registration/Deregistration Use-Case

The first step a user has to accomplish, in order to use an IMS network, is the registration. At registration, the user informs the IMS network about its location and establishes a secure communication channel for later signalling between the UE and the network. This information is stored in the home domain registrar and used by the network entities to route terminating messages toward it. De-registration is the operation to remove the registered contact information.

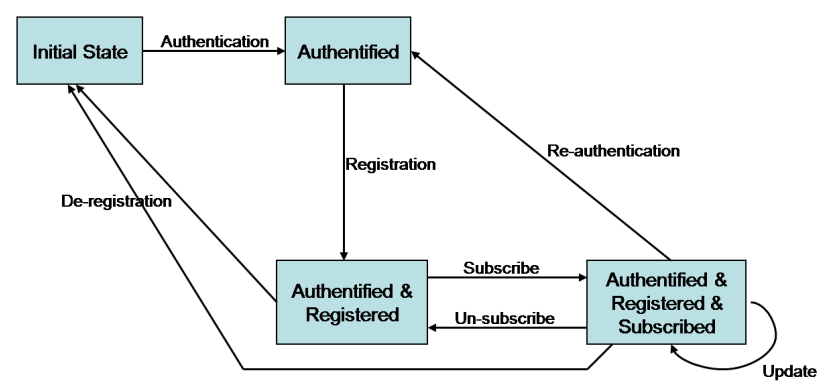

Figure 5.4: Registration State Machine

Figure 5.4 presents the steps a user executes during the registration process. The user starts from an *initial state* where the user is connected to the IP network and can start the authentication procedure. The first step to register is the *authentication* when the P-CSCF negotiates a set of security associations with the UE. Future registration/deregistration operations performed over these secure channels do not need to further be authenticated as the integrity of the messages is protected.

Once authenticated, the user may start the registration procedure. The registration has attached also an expiration timer which indicates for how long the user will remain registered. After registration, the user will subscribe to its own registration status in order to be able to react to network initiated events regarding its registration status. After subscription, the user will receive updates on changes of its status, e.g., the user will be deregistered sooner than it required. After registration, the user may start anytime the deregistration procedure thus going back to the initial state.

The state machine opens the perspective for many scenarios definition. However, only a few of them which cover some of the most representative paths are selected.

5.4.1.1 Scenario 1.1: Initial Registration

This scenario includes the transitions from initial state to authentication, registration and subscription. Each user is forced to execute this flow before starting using any other IMS service. The call flow for this scenario is presented in Figure 5.5. The user starts the registration procedure by sending a REGISTER message, requiring the IMS network to send back an authentication key. The 401 message indicates that the user is asked to authenticate. In response, the user computes the response and issues a new REGISTER message (this time including the response). If everything works well, the networks response with a `200 OK`.

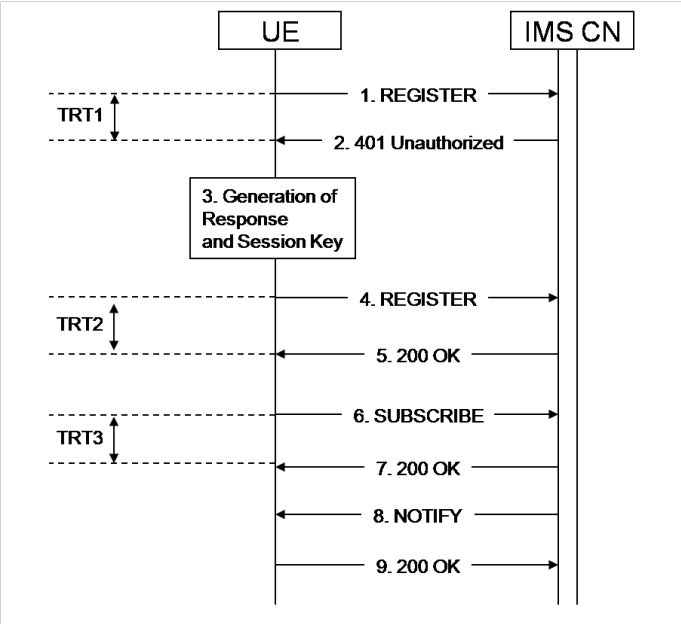

Figure 5.5: Sequence of Messages for Initial Registration Scenario

5.4.1.2 Scenario 1.2: Re-Registration

Any registration has an expiration time. When the validity of a registration is about to expire, the user has to renew its registration. To simulate this behaviour, the re-registration scenario has been defined. The re-registration call flow is presented in Figure 5.6. It consists only of two messages which are `REGISTER` and its `200 OK` response. This time the user does not have to authenticate again, since its credentials are still valid.

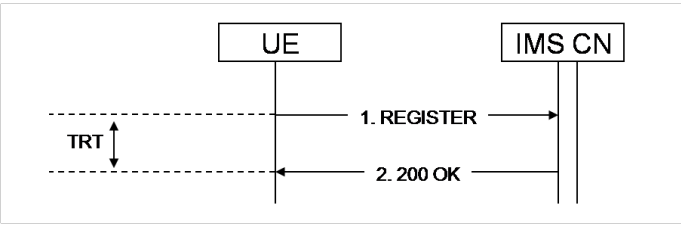

Figure 5.6: Sequence of Messages for Re-Registration Scenario

5.4.1.3 Scenario 1.3: Re-Subscription

Similar to registration, the subscription expires after some time. In general, the subscription and registration have different expiration times, therefore, a re-subscription is also needed from time to time. The call flow for the re-subscription scenario is illustrated in Figure 5.7.

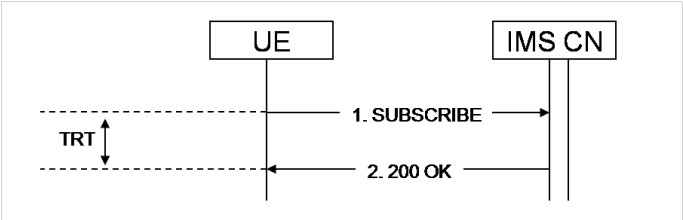

Figure 5.7: Sequence of Messages for Re-Subscribe Scenario

5.4.1.4 Scenario 1.4: De-Registration

The deregistration happens when a user switches off its device. This scenario occurs in similar proportions as the initial registration since after a deregistration, usually an initial registration takes place. A typical example is when the user switches off its device in the plane before the take off, and initiates a new registration after landing. The deregistration is similar to re-registration call flow with the remark that the expiration time is zero.

5.4.1.5 Scenario 1.5: Failed Initial Registration

A typical fail scenario is the user-not-found case. It may happen when a user tries to register to a new network being a non valid subscriber. In this case the IMS network is expected to respond with a “404 Not Found”.

5.4.2 Session Set-Up/Tear-Down Use-Case

This use-case regards the establishment of a multimedia session, e.g., voice call, between two registered users. To be realistic enough, during the set-up period a "ringing" delay is introduced. Both steps of the session are concerned: session set-up and the session tear-down. Therefore, between the two steps, a talking time is foreseen. The most common situations encountered in real live are:

- *Call Successful* - when everything goes right, the session is successfully established but also successfully terminated.
- *Call Abandoned* - when the originating party cancels the call during ringing, before the terminating party answers.
- *Call Rejected* - when the terminating party rejects the call while busy.
- *Call Failed* - when some information in the call establishment is wrong and causes a failure, e.g., terminating party not found.

In the real-world, these kinds of communication situations occur with a certain frequency, so they must be included in the traffic set of the benchmark test scenarios, in order to make it more realistic.

An important aspect is the resource reservation (of video and audio streams) status on the two participating sides. During the call establishment, any party might require resource reservation,

thus the call flow will be more complex for these scenarios. The resource reservation can be multiplexed with the first three situations described above: successful, abandoned and rejected call.

Moreover, there are several waiting times during which the Test System must pause, like the *ringing time* or the *call holding time*. Distribution of these delays can follow a constant or a Poisson distribution. During the set-up period, a "ringing" delay is introduced and if the set-up is successful then the scenario is put on hold for the duration of the call and then it is terminated with a tear-down sequence. After the "talking" time, one of the users issues a `BYE` message to close the session. The `BYE` is answered by an `OK` confirmation message.

In the case of the calling scenario some metrics are measured, including: the delay for invite request, the delay between consecutive invite requests, the delay of established calls and so on.

5.4.2.1 Scenarios 2.1-2.4: Successful Calls

Scenario 2.1: Successful Call without Resource Reservation

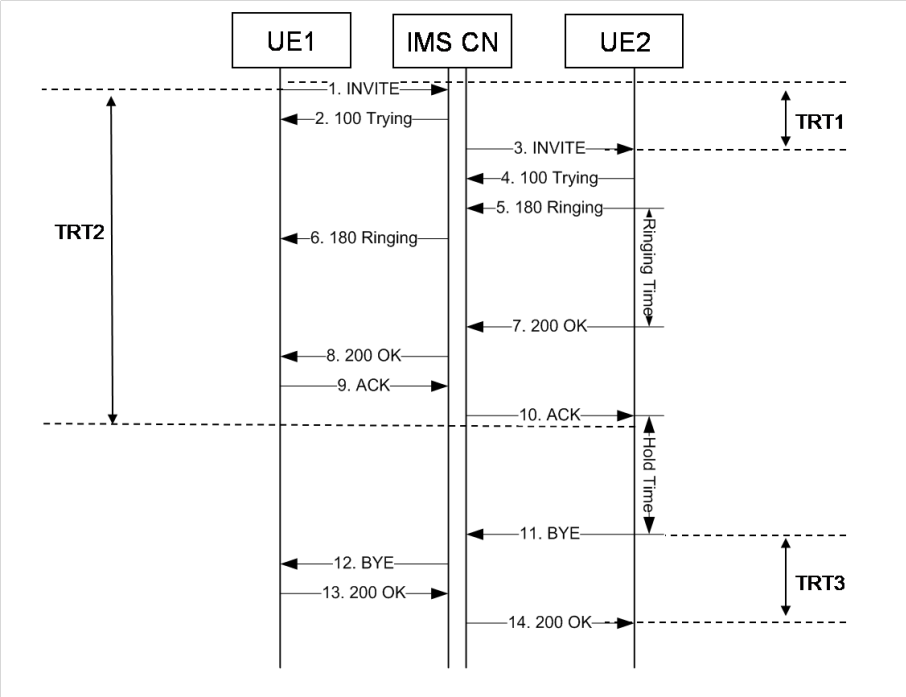

Figure 5.8: Sequence of Messages for Successful Call without Resource Reservation Scenario

Figure 5.8 depicts the sequence of messages of a voice call. The call scenario implies two UEs which establish a call session. Each message is indicated in the message flow by a name and a sequence number according to its order in the flow. One of them plays the role of caller by sending the `INVITE` message (message 1) to the IMS network and the other one plays the role of callee by accepting the call invitation (message 3). The first 10 interchanged messages between the two entities are used for establishing the connection only. The UE2 first answers with `100 Trying` (message 4) to announce the IMS network that the `INVITE` has been received and that it tries to reserve resources for the call. After the resource reservation step, the UE2 sends a `180 Ringing`

message (message 6) to announce the UE1 that the UE2's phone is ringing. When the UE2 answers the call, a `200 OK` message (messages 7 and 8) is sent to UE1. The UE1 confirms the call establishment with an `ACK` message (messages 9 and 10). After a *talking time* period, one of the UEs decides to close the call by issuing a `BYE` message (messages 11 and 12). The second UE confirms the call finalisation with a `200 OK` (messages 13 and 14).

For this scenario, a number of metrics are interesting. The TRT1 is introduced to measure the time required by the network to route the `INVITE` message from the originating party to the terminating party. The TRT2 measure the session tear-up part and indicates the time required by the network to establish a call. Similarly, the TRT3 measures the session tear-down part.

Scenario 2.2: Successful Call with Resource Reservation on Both Sides.

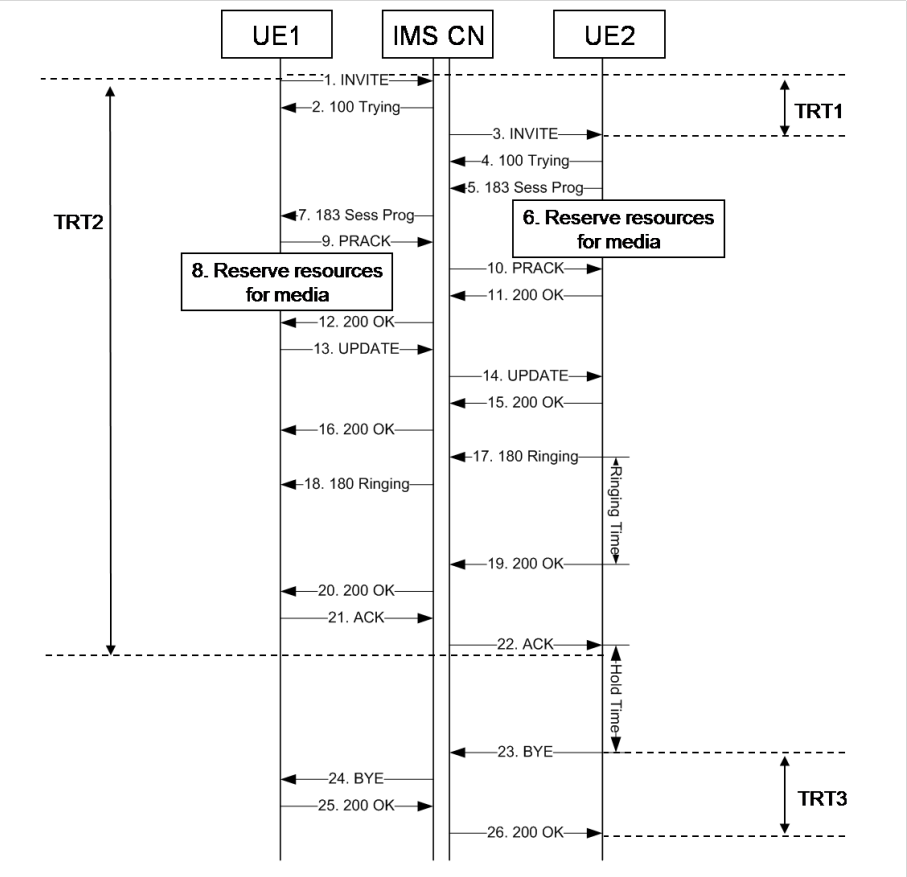

Figure 5.9: Sequence of Messages for Successful Call with Resource Reservation on Both Sides Scenario

In this scenario, both of the UEs have to complete the resource reservation. The `UE1` completes resource reservation shortly after the arrival of the `183` response. The terminating UE completes local resource reservation after sending a provisional response to the `INVITE` request. As soon as the `UE1` has finished the resource reservation, it acknowledges the `183` response with a `PRACK` request. Resource reservation at `UE1` is assumed to have completed at some point prior to sending the PRACK request. The `200 OK` response sent by the UE2, lets the UE1 that the `PRACK` has been received. When the resource reservation is completed, `UE1` sends the `UPDATE` request to the terminating endpoint. After the confirmation of the `UPDATE`, the rest of the call flow looks similar

to successful call without reservation scenario.

Also for this scenario, the metrics defined for the successful call without reservation can be applied. However, this time it is expected that the TRT2 takes far more time than for the simple successful call.

Scenario 2.3: Successful Call with Resource Reservation on Origination Side

In this case only the originating party needs to reserve resources as the terminating one is considered to have them already. The call flow looks identical to the call flow of successful call with resource reservation on both sides, with the difference that the step 6 "Resource reservation for media" is missing for the UE2.

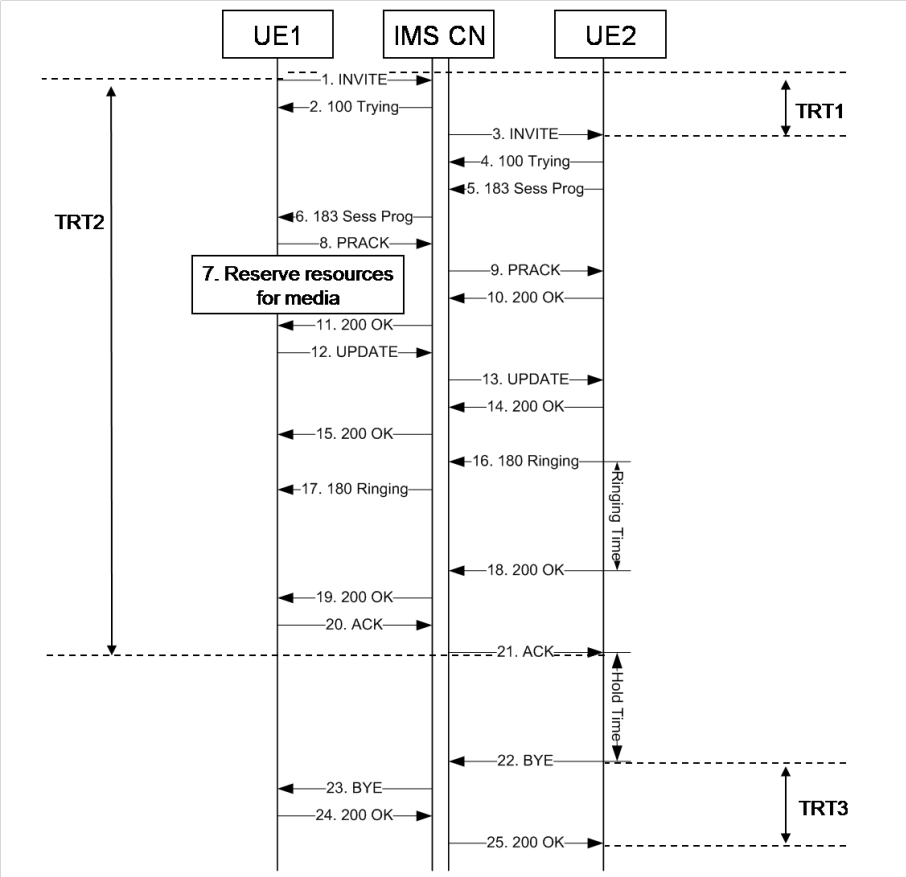

Figure 5.10: Sequence of Messages for Successful Call with Resource Reservation on Originating Side Scenario

Scenario 2.4: Successful Call with Resource Reservation on Terminating Side.

In this case only the terminating party needs to reserve resources. The call flow looks identical to the call flow of successful call without resource reservation on both sides, with the difference that the step 5 "Resource reservation for media" is introduced on UE2 side to make the resource reservation.

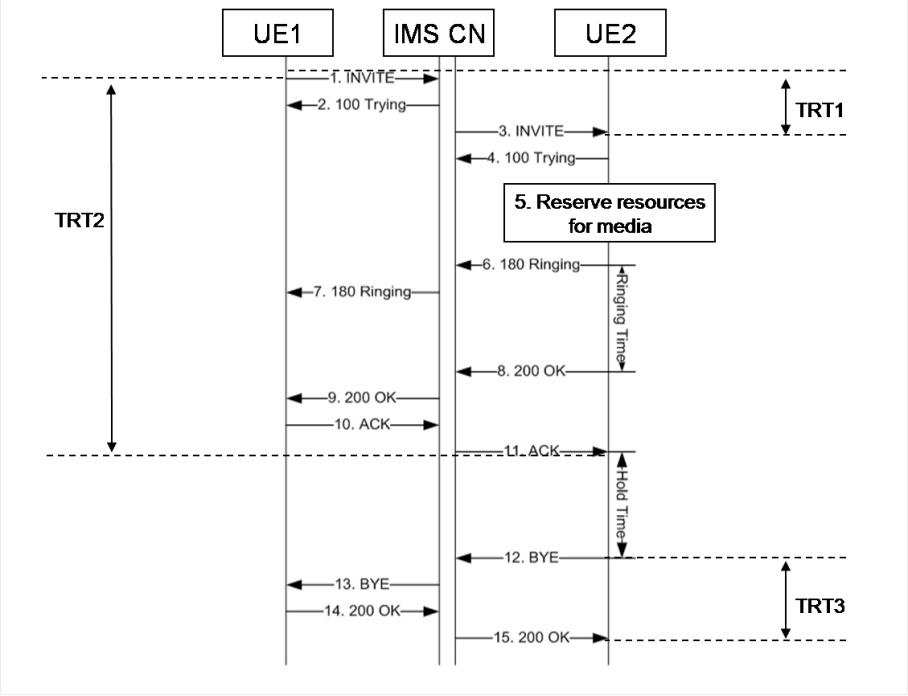

Figure 5.11: Sequence of Messages for Successful Call with Resource Reservation on Terminating Side Scenario

5.4.2.2 Scenarios 2.5-2.8: Abandoned Calls

In these scenarios, the terminating user does not answer the phone during the ringing time and the originating user abandons the call. The first part of the call, is similar to the successful call scenarios for all combinations with respect to resource reservation. The termination, which is different from the successful call scenarios, is illustrated in Figure 5.12. The termination is the same for all abandoned scenarios with or without resource reservation and consists of a `CANCEL` transaction. The originating party sends the `CANCEL` message which is quickly answered by the IMS network without waiting for a reaction from the terminating party. The `CANCEL` is sent by the IMS network also to the terminating party which has to answer it with a `200 OK`. Additionally, the terminating party sends also a `487` Request Terminated message which is answered by the network with an `ACK` message. The network sends the `487` further to the originating party which has to replay with an `ACK`. The time between the `CANCEL` message until the receiving of `487` can help to measure how long it takes the IMS network to process the abandon sequence. The `TRT3` has been introduced to measure this time.

The complete list of abandoned scenarios, taking into account the resource reservation, is:

Scenario 2.5: Abandoned Call Scenario without Resource Reservation

Scenario 2.6: Abandoned Call Scenario with Resource Reservation on Both Sides

Scenario 2.7: Abandoned Call Scenario with Resource Reservation on Origination Side

Scenario 2.8: Abandoned Call Scenario with Resource Reservation on Terminating Side

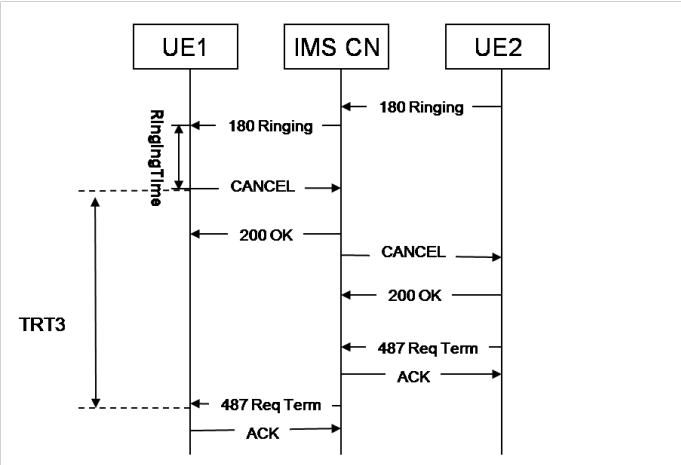

Figure 5.12: Sequence of Messages for Abandoned Termination Scenario

5.4.2.3 Scenarios 2.9-2.12 Rejected Calls

These scenarios illustrate the case when the terminating user refuses the call after the ringing time. The first part of the call including the resource reservation is similar to the successful call scenarios.

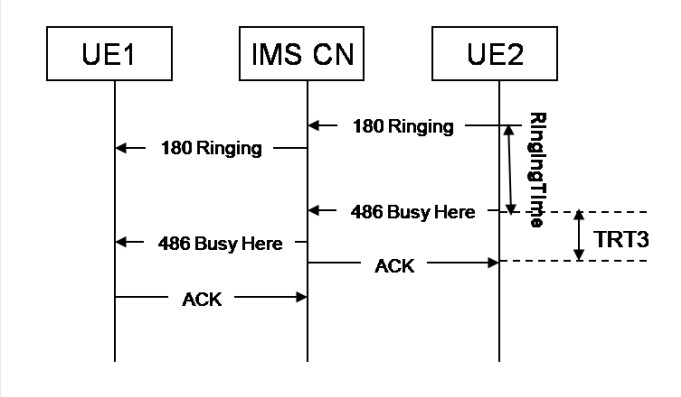

Figure 5.13: Sequence of Messages for Rejected Termination Scenario

The complete list of rejected scenarios, taking into account the resource reservation, is:

Scenario 2.9: Rejected Call Scenario without Resource Reservation

Scenario 2.10: Rejected Call Scenario with Resource Reservation on Both Sides

Scenario 2.11: Rejected Call Scenario with Resource Reservation on Origination Side

Scenario 2.12: Rejected Call Scenario with Resource Reservation on Terminating Side

5.4.2.4 Scenario 2.13: Failed Calls

A call failure occurs when some information in the call establishment is wrong. When the call cannot be created, the SUT responds with an error message. A simple scenario is, for example, when terminating party is not found. In this scenario, the SUT responds with the `404 Not Found` message (see Figure 5.14).

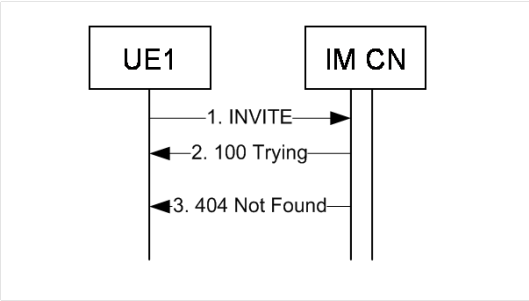

Figure 5.14: Sequence of Messages for Call Fail Scenario

5.4.3 Page-Mode Messaging Use-Case

This service is a simple data transport service between two users. The service has a limitation for the size of the messages but it is an ideal service for transmitting short messages between users thanks to the short call flow which consists of only two messages.

Though this service cannot harm too much the performance of the IMS network, it should also be included in the traffic set since many users utilise it quite frequently.

Since the service consists only of two messages: one request and one confirmation, only the successful (S3.1) and the fail (S3.2) scenarios are identified.

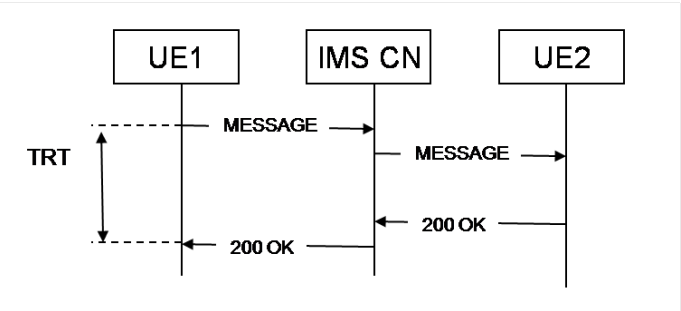

Figure 5.15: Sequence of Messages for Page-Mode Messaging Scenario

5.4.4 Use Case Representativeness

The benchmark traffic set consists of scenarios which belong to the most common IMS use cases that are encountered the most in the real life deployments. As argued earlier in this thesis, the

representativeness of the selected scenarios has a considerable impact on the significance of the produced results. One simple rule is that the more complex the scenarios are, the more significant the results are. The representativeness of the chosen use cases is evaluated considering the following arguments:

- *SIP protocol coverage* - overall, any type of Request and Response defined by the SIP protocol is included in at least one scenario. This ensures a good coverage of the SIP protocol with respect to test data types.
- *IMS application coverage* - IMS supports various types of applications: voice call, conference, IPTV [Har07], presence, Facebook, etc. All these applications rely on the three recognised use-cases: registration, voice-call and messaging. These types of interactions are recognised in any other application. Therefore, the selected use cases ensure also a good coverage with respect to the interactions types between UEs and IMS network.
- *negative testing* - Not only the positive interactions are treated, e.g., successful calls, but also negative situations, e.g., abandoned, rejected calls. This ensures a good coverage of typical user behaviours as encountered in reality.
- *risk management* - The three selected use cases lay as basis for further types of interactions, e.g., conference, IPTV. Therefore, any IMS network provider should ensure a good functionality of the basic types of interactions.

5.5 Tools Specialised for IMS Testing

As long as the case study presented in this thesis regards the SIP [IET05] protocol, highlighting on some of the available tools is necessary.

5.5.1 SipStone

The SipStone [SNLD02] is an example of realising a performance test by using a simple and specific architecture. It is a benchmark for SIP which is designed to be a benchmark with a single standardised implementation and workload. The implementation performs a series of tests that generate the pre-configured workload. This workload simulates the activities of multiple users initiating SIP calls. The workload is intended to be simple and repeatable, and satisfy workload requirements such as concurrent initiation of calls by multiple users, random call arrivals and the forwarding of requests.

SipStone implementation is a tool which consists of three components: Controller, Loader, and Callhandler. The controller is the master program that starts request generators to generate SIP requests. It uses loaders for request generation and call handlers for responding to generated calls. The Loader emulates a SIP User Agent Client (UAC) and is the request generator. The Callhandler emulates a SIP User Agent Server (UAS) and runs the actions of receiving SIP requests, and responding according to the protocol sequence.

The benchmark consists of a set of SIPstone load generators that create the SIP request load, a call handler that simulates a user agent server and a central benchmark manager that coordinates the

execution of the benchmark, and the SUT. The call handlers may run along with the load generators or on different systems; the distribution is based on mechanisms such as rsh/rcmd [SN90] or Java Remote Method Invocation (RMI).

SIP requests for the benchmark are generated by UACs. Depending on the request rate needed, one or more clients may be needed to generate the requests. The requests are directed to the UASs which take the appropriate protocol actions.

5.5.2 SIPp

SIPp [JG06] is a SIP test tool and traffic generator. Sponsored by HP, its source code is open sourced and at the moment it enjoys the position of the fastest and most flexible SIP test tool. It is a tool specific for testing the SIP protocol and it efficiently uses the advantage of this constraint.

SIPp is capable of generating thousands of calls per second, it can also generate any SIP scenario. By default it includes several SIPStone scenarios and complex call flows can easily be generated through comprehensive XML [W3C08] configuration files. It features dynamic displays of statistical data as well as periodic dumps for further processing and user interactive traffic rates.

The first phase in a test specification is the definition of the test interfaces. This consists of XML dictionaries on top of the base protocols that define each particular reference point used in the test. Then the scenarios for tests are described in other XML files and the test is run in an interactive interface that shows a set of dynamically computed metrics and also allows for user interactions through commands. One thing that these test tools lack is the concept of subscriber or user to perform a set of specific scenarios. In order to achieve this functionality the total traffic that the subscribers would generate has to be simulated as an aggregation of scenarios.

Load generation is achieved by starting in parallel the scenarios that constitute the traffic mix, with the given individual call rates required.

5.5.3 IxVoice

IxVoice [Ixi07] service is a VoIP [DPB+06] test platform to provide both functional and bulk VoIP testing. Ixia's VoIP test solution provides an environment of real-world triple-play traffic that accurately models live network environments. Using Ixia test platform, a single test chassis emulates millions of realistic call scenarios. IxVoice SIP emulates user agents that use SIP and Real-Time Transport Protocol (RTP). Default configurations and state machines can be constructed for any session lifetime scenario.

5.5.4 Spirent Protocol Tester

Spirent's IMS solution [Spi06] is based on a customisable state-machine with high performance and scalability. The tool can test or emulate any IMS core network element including CSCFs, HSS, AS. Additionally, multiple protocols, e.g., SIP, Diameter, RTP, can be used simultaneously in the test process. With respect to the test scenarios, the tool can run custom call flows at a high load rate. Technically, the workload is realised by running the same test cases used to perform functional testing in parallel.

5.6 Benchmark Specification in TTCN-3

TTCN-3 is used to specify the behaviour of the benchmark scenarios and the load characteristics, e.g., number of scenario attempts per second, number of users, benchmarking time. In this section the TTCN-3 language elements used to implement the benchmark specification are explained.

5.6.1 Test Components

In TTCN-3, the test component is the building block to be used in order to simulate concurrent user behaviours. The parallelism is realised by running a number of test components in parallel. For better performance, the test components are distributed over several hosts by using an execution framework which implements the TCI interfaces as described in Chapter 4.

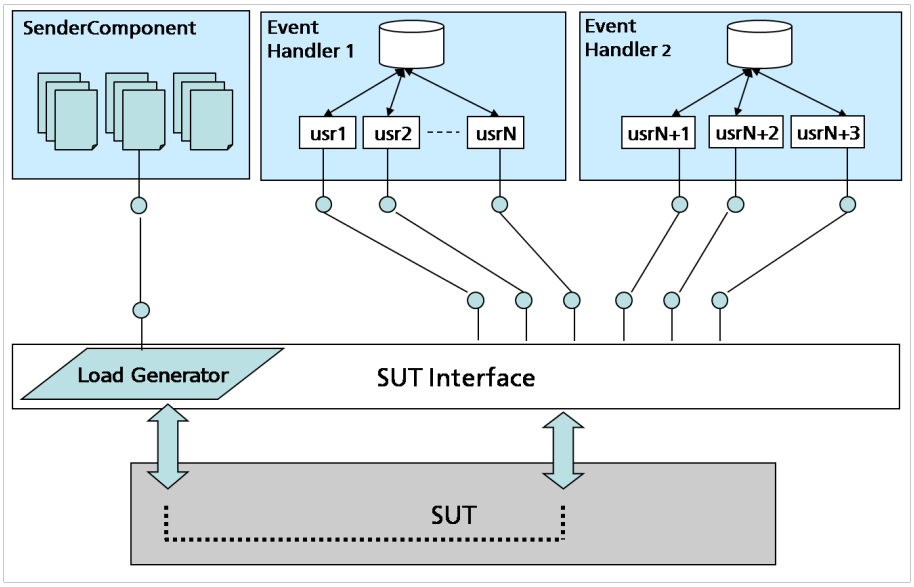

Figure 5.16: Test System Configuration

The test specification consists of a collection of test components that emulate the users of the IMS system, i.e., the SUT. The test components interact with the SUT via TTCN-3 `send` and `receive` statements. For performance reasons, the User Handler with Interleaved Users per Thread (UH_InterleavedUPT) pattern described in Section 3.5.2 is applied. In this pattern, the behaviour of each test component simulates a number of parallel user behaviours at the same time. The test component may work concurrently on many transactions at the same time. This way, the number of parallel threads running on one machine is reduced and, consequently, less CPU is consumed for thread context switches (a well known performance problem).

The test configuration is presented in Figure 5.16. The load generation is realised by an external process instantiated in the adaptor layer. The applied pattern for load generation is Load Generation with Multiple Generators and Decentralised Data (LG_MGenDectrl) defined in Section 3.5.6. Therefore, each test node has its own load generation process. Each process is controlled by a test component of type `EventSender`, which creates template messages for the initial protocol messages meant to create calls, e.g., `INVITE`. This approach is chosen in order to improve the precision of the load generator. The project required that the calls are sent to the SUT following a Poisson

distribution which requires a precision of one millisecond. The implementation in TTCN-3 proved to be very resource consuming and lacked precision. Therefore it has been decided to move the load generator in the adaptor. The templates are used by the load generator process to initiate calls; a template instance is bound to a specific user and it is reused anytime the load generator creates a call for that user. This approach encapsulates the generation process allowing the use of third party load generators. The TTCN-3 component is used in this approach only to configure the external process with the message templates that need to be sent to the SUT.

Each call created by the load generator is associated to an `EventHandler` component, which will handle all required transactions for that call. The event handling behaviour is run on the `EventHandler` component type. The number of `EventHandlers` is arbitrary and depends on the number of simulated users and on the performance of the hardware running the test system. Typically, an `EventHandler` simulates a few hundreds of users.

5.6.2 Test Distribution

The test distribution is realised at the test component level, therefore the test tool will instantiate on each test node a number of test components. The distribution is configured as shown in Listing 5.1.

Listing 5.1: Test Component Distribution

```
<?xml version="1.0" encoding="UTF-8"?>
<componentassembly xmlns="http://www.testingtech.de/xsd/ttcn/tcdl"
                   xmlns:tcdl="http://www.testingtech.de/xsd/ttcn/tcdl"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.testingtech.de/xsd/ttcn/tcdl
                   com/testintech/ttcn/platform/xml/tcdl.xsd">
  <description>A distributed test</description> <special container="host1" />
  <partition>
    <component_selectors>
      <componenttype>EventSender</componenttype>
    </component_selectors>
    <homes distribution="round-robin">
     <container id="host1" /> <container id="host2" />
     <container id="host3" /> <container id="host4" />
    </homes>
  </partition>
  <partition>
    <component_selectors>
      <componenttype>EventHandler</componenttype>
    </component_selectors>
    <homes distribution="round-robin">
         <container id="host1" /> <container id="host2" />
         <container id="host3" /> <container id="host4" />
    </homes>
  </partition>
  <partition>
    <component_selectors>
      <componenttype>LoadController</componenttype>
    </component_selectors>
    <homes>
      <container id="host1" />
    </homes>
  </partition>
  <collector>
```

```
    <container id="host1" />
  </collector>
</componentassembly>
```

In this configuration, one `EventSender` component is created on each test node. Listing 5.2 presents the TTCN-3 code which creates the `EventSender` components. When a new component of this type is created, the execution environment investigates the distribution rules given in the XML file. The **for** loop creates `PX_NumberOfServers` components which are distributed by using the round-robin algorithm. Therefore, in order to have a single `EventSender` per test node, the `PX_NumberOfServers` parameter has to be equal to the number of test nodes.

Listing 5.2: EventSender Creation

```
for (var integer i := 0; i < PX_NumberOfServers; i := i + 1) {
  eventSender[i] := EventSender.create;
  connect (self:p2EventSender[i], eventSender[i]: p2LoadController);
  eventSender[i].start (userMainHandler(i));
}
```

In order to handle the local created transactions several `EventHandler` components are created on each server. The `EventHandlers` can handle only transactions created by the `EventSender` on the same test node. The handling of "remote" transactions would be possible, but with significantly more overhead. The TTCN-3 code which creates the `EventHandler` components is presented in Listing 5.3. The **for** loop creates `NR_OF_COMPONENTS*PX_NumberOfServers` components which are distributed according to the rules given in the XML file. For homogeneous test hardware configurations an equal number of components will be distributed on each node. Therefore, the `PX_NumberOfServers` is equal to the number of test nodes involved in the test execution. The `NR_OF_COMPONENTS` is the number of `EventHandlers` per test node and it is such selected that the parallelism of a test node is increased. It should be at least equal to the number of CPUs on a test node so that each CPU is used to run an `EventHandler` process. However, the `NR_OF_COMPONENTS` parameter can be also determined empirically through experiments in order to find out the best performance of the test system.

Listing 5.3: EventHandler Creation

```
var EventHandler eventHandler[NR_OF_COMPONENTS*PX_NumberOfServers];
for (var integer i := 0; i < NR_OF_COMPONENTS*PX_NumberOfServers; i := i + 1) {
  eventHandler[i] := EventHandler.create;
  connect (self:p2EventHandler[i], eventHandler[i]: p2LoadController);
  eventHandler[i].start (userStateHandler());
  TSync.start; p2EventHandler[i].receive(SYNC); TSync.stop;
}
```

5.6.3 Event Handling

Figure 5.17 illustrates the behaviour of an `EventHandler` component. The implementation of the `EventHandler` is based on the State Machine with Generic Handler (SM_GenHdl) pattern presented in Section 3.5.1. An event handler processes events received from the SUT and executes appropriate actions according to the scenario message flow (as described in Section 5.4). The event processing starts with the *identification* of the `user_id` for which the message is received. This information is extracted from the protocol information embedded in the message. Once the user is identified, the handler evaluates the current state of that user and *validates* if the new message

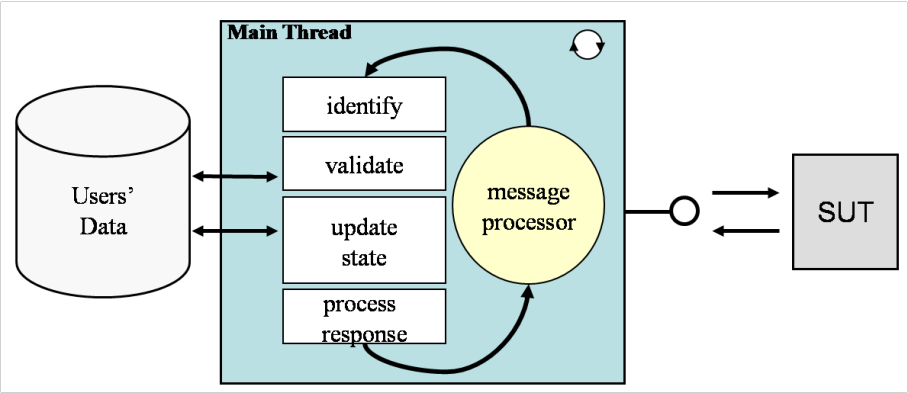

Figure 5.17: Event Handler

corresponds to a valid state, otherwise the transaction is considered *inadequately handled scenario attempt*. Next, the user state is *updated* in the local user information database (accessed over TTCN-3 external functions). If the received message, requires follow-up actions on the test system side, new messages are created and sent to the SUT. When receiving or sending any message, a log event is generated with precise timestamp for the evaluation of the SUT latency.

Listing 5.4: State Processing

```
alt {
  // P-CSCF --INVITE-> User
  [] p2SUT.receive(INV_Req_r)->
     value invReq {
  }
  // User <- 200 OK - P-CSCF
  [] p2SUT.receive(Resp_200_r_INV)->
     value response {
  }
  // P-CSCF --ACK -> User
  [] p2SUT.receive(ACK_Req_r)->
     value ackReq {
  }
  // User <- BYE -- P-CSCF
  [] p2SUT.receive(BYE_Req_r)->
     value byeReq {
  }
  // User <- 200 OK -- P-CSCF
  [] p2SUT.receive(Resp_200_r_BYE)->
     value resp {
  }
  // User <- 408 -- P-CSCF
  [] p2SUT.receive(Resp_408_r)->
     value resp {
  }
  // User <- 480 -- P-CSCF
  [] p2SUT.receive(Resp_480_r)->
     value resp {
  }
  // User <-- ? response --P-CSCF
  [] p2SUT.receive(Resp:?) {
  }
```

```
    // User <-- ? request --P-CSCF
    [] p2SUT.receive(Req:?) {
    }
}
```

The event handling process is implemented by using the `alt` construct which defines an alternative for each possible message type (see Listing 5.4). The alternatives capture expected message types in accordance with the implemented scenario, e.g., `INVITE, 200 OK, ACK, BYE`, but also unexpected message types which might occur when the scenarios are not handled correctly by the SUT, e.g., `408, 480`. However, the receiveing of any other unexpected message is foreseen by using the `?` template.

5.6.4 Protocol Messages

Listing 5.5: SIP protocol messages specified in TTCN-3

```
// general type for a SIP Request
// can be extended by any
// other request type (e.g. INV_Req)
type record Request {
    integer transactionId,
    RequestLine requestLine,
    MessageHeader msgHeader,
    charstring messageBody optional
}
```

The specification of the SIP message types derives from the ETSI type system for SIP conformance testing [ETS05], which is consistent with the SIP protocol specification [IET05].

SIP is a text-based protocol that allows structured presentations of the same information. A TTCN-3 implementation of SIP should describe the protocol messages into TTCN-3 type structures and values. The instances of these types are encoded into the textual representation. The received messages are decoded back into the abstract presentation format.

The top message type is `SIP_Request` (see Listing 5.5), which contains a `requestLine`, a `messageHeader` and a `messageBody`. An additional field which is `transactionId` is used to track the SIP transactions. Except `transactionId`, all other fields are of structured types such as the `MessageHeader` type presented in Listing 5.6. The structure of the SIP message types is refined up to leafs fields of basic types such as `integer`, `charstring`, `bitstring`. An example of a terminal element is the `Authorization` type presented in Listing 5.7.

Listing 5.6: The MessageHeader type

```
type set MessageHeader {
    Authorization authorization optional,
    CallId callId optional,
    Contact contact optional,
    CSeq cSeq optional,
    Expires expires optional,
    From fromField optional,
    RecordRoute recordRoute optional,
    Route route optional,
    ServiceRoute serviceroute optional,
    To toField optional,
    Via via optional,
```

```
    MaxForwards maxForwards optional,
    ContentLength contentLength optional,
    WwwAuthenticate wwwAuth optional,
    Event event optional
}
```

Listing 5.7: The Authorization type

```
type record Authorization {
    FieldName fieldName(AUTHORIZATION_E),
    //Credentials body
    charstring body optional
}
```

5.6.5 User State Processing

Listing 5.8: Example of handling Resp_200_OK_INV event

```
[] p2SUT.receive (Resp_200_OK_INV)->
    value resp {

    // create an ACK request
    ackReq := ACK_Request_s;

    // create a new transaction
    ackReq.transactionId := getNewTrId();

    // use the same callId and seqNumber
    ackReq.msgHeader.callId :=
        resp.msgHeader.callId;
    ackReq.msgHeader.cSeq.seqNumber :=
        resp.msgHeader.cSeq.seqNumber;

    // set from, to, via and route
    ackReq.msgHeader.fromField :=
        resp.msgHeader.fromField;
    ackReq.msgHeader.toField :=
        resp.msgHeader.toField;
    ackReq.msgHeader.via :=
        resp.msgHeader.via;
    ackReq.msgHeader.route :=
        getServiceRoute(resp);

    // send the ACK
    userChannel := getChannel(resp);
    p2SUT.send (ackReq) to userChannel;

    // close the transaction
    delTransactionId(resp.transactionId);
}
```

SIP protocol messages are bound to carry state-full information which has to be correlated with information retrieved from other messages. In a state-full testing approach, the test behaviour defines a sequence of requests and settings for controlling the state maintained between them. Most performance testing tools support session tracking mechanisms based on unique session identifiers.

State-less workloads do not require to keep track of previous messages and, therefore, they are easier to simulate. State-less workloads tools are usually called traffic generators and have the purpose to only send requests to the SUT, following a given pattern. However, these tools do not validate or react to SUT's responses.

The implementation is based on the state-full approach which helps controlling the whole flow of actions specified in the test scenario. The state information is maintained in a Java [SM08] hash object created in the adaptor, but controlled from TTCN-3 over external functions.

The state processing is exemplified in Listing 5.8. In that example, a `200 OK` SIP message (see Figure 5.8, message 8) is received by the test system. The message matches the `Resp_200_OK_INV` template and it is decoded in the `resp` variable. According to the scenario message flow (see Figure 5.8, message 9), an `ACK` message should be sent back to the SUT. This message is created from the `ACK_Request_s` template. Some fields of the `ACK` template are already set by default but most of them have to be set with information retrieved from the `Resp_200_OK_INV` message. The new message is then sent back to the SUT by using the `p2SUT` port. Since the component has only one port connection to the SUT, the correct identification of the user, which has to receive the message, is realised by using the `userChannel` variable of type `address`.

5.6.6 Test Adaptor

The connections to the SUT are implemented via ports and an adaptation layer which handles the SIP messages interchanged with the SUT.

The adaptation layer is based on the National Institute of Standards and Technology (NIST) Jain SIP [OR05], a Java-standard interface to a SIP signalling stack, and it implements the TRI interface [ETS07b] for TTCN-3 test adaptation. It provides a standardised interface for handling the SIP events and event semantics, and offers full transaction support for SIP-based calls. The relation between the test adaptor, SIP stack and the SUT is depicted in Figure 5.18. The test adaptor registers an implementation of the SipListener interface to interact with the SIP Stack. The `triSend(...)` operation [ETS07b] uses stack objects via the SipProvider in order to create or use protocol transactions. The test adaptor receives back messages from the stack as Events, via the SipListener interface.

The NIST stack has been modified to support multiple listening points (one per user) and to improve the performance of the I/O handling [Pet06]. Initially, the stack had used one thread instance for each request. This model is not adequate to generate large amount of requests due to the overhead to create new threads for each request. It also consumes a large number of operating system resources in order to handle context switches between many threads.

In the redesign, the thread pool concept is applied. It uses a fixed number of N threads to handle a big number of M tasks (N$\ll$M). A thread can handle only one task at a time, but as soon as the thread finishes its task, a new task can be processed. The awaiting tasks are kept in a queue wherefrom any thread may pull tasks.

The test logic, i.e., state handling, is executed by the TTCN-3 parallel test components. In order to maintain the association between user transactions, created by the SIP Provider, and the corresponding `EventHandlers` at TTCN-3 level, a Java HashMap object is used. The HashMap contains value pairs of transaction IDs as those created by the SIP Provider and TTCN-3 message instances identified by the `transactionId` field (see Listing 5.5). The HashMap is updated whenever a new transaction is created or closed.

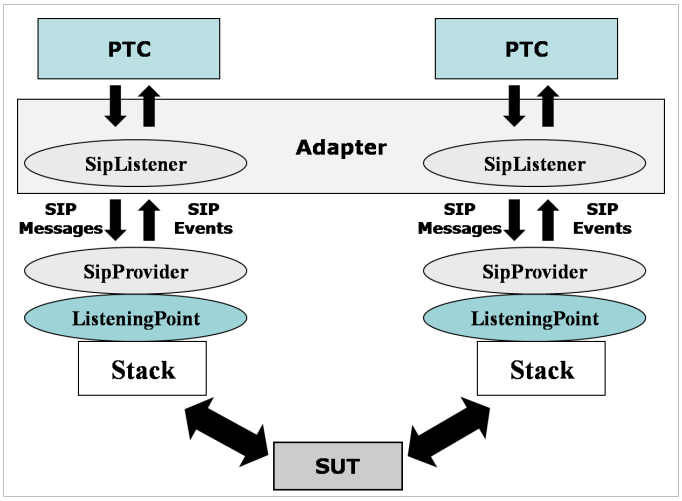

Figure 5.18: Test Adaptor Architecture

5.6.7 Data Encoding

The data is seen in TTCN-3 at an abstract level, but during execution it has to be encoded into protocol messages, which are then transmitted over the network layer. Usually, a higher flexibility at data specification level affects more the performance of the tool since more CPU is required to transform data to a lower level.

In the implementation, the type system is based on ETSI SIP Test Suite defined in [ETS05]. This representation offers flexibility, allowing a detailed manipulation of the SIP messages. However, for improving the performance, some of the fields currently implemented as structured types may be simply transformed into **charstring**.

5.6.8 Technical Details

TTCN-3 proved to be an easy-to-use language. But, along with the easy-to-use also comes more overhead and resource load on the test nodes and it becomes even more important to look at the performance issues. Therefore, a number of technical performance improvements [Shi03] were needed:

- the load generator has been moved in the adaptor in order to increase the precision of the event distribution as explained in Section 5.6.
- the load generation has been separated from the event handling. The initial message of each call is sent by only one component of type `EventSender`. Then, all the SUT responses are handled by the `EventHandlers`.
- the simulation of more than one user per test component instead of one single user per test component is far more efficient. In this approach, the behaviour was a little more complex to specify since a mechanism to find out for each message to which user it belongs is required.
- the adaptor has been modified to use the more efficient concept of pools of threads instead of having a new thread for each transaction.

- the offline logging processing has been used instead of real-time logging in order to avoid additional processing operations.

Another problem, was the garbage collection which affected seriously the precision of the test system actions [BCM04](the tool is developed with Java [SM08] technology). The Java virtual machine calls the garbage collector from time to time to release unused memory. For higher loads, the garbage collection happens a few times per second. During the garbage collection, the Java virtual machine stops all threads currently running. Moving the load generator from TTCN-3 to the adaptor improved the precision. However, for even better precision, the platform should move to a real-time Java virtual machine which offers the possibility to create threads which are not stopped during the garbage collection.

5.7 Experiments

The experiments presented in this section serve two major goals. The first goal is to show a concrete execution of the IMS benchmark with a complete analysis of the performance of the tested SUT. Additionally, the benchmark is applied to different hardware configurations in order to illustrate how the test can be used for performance comparison. The second goal is to discuss in more detail some of the test parameters which influence the results. It is important to understand them and learn how to select valid values for them such that the results are also valid and meaningful for the performance analysis.

5.7.1 Testbed Environment

The hardware used to run the IMS Benchmark test system consists of multiple rack-mount servers providing enough processing power to produce the requested loads. Depending on the requested load for a particular test, up to 8 test servers are currently used. The current configuration consists of 5 HP DL380 G4 (dual Dual-Core Intel Xeon Processor 2.80 GHz, 2*2M Cache, 800 MHz FSB, 12GB memory) and 3 Supermicro rack-mount servers (dual Dual-Core Intel Xeon Processor 5150, 4M Cache, 2.66 GHz, 1333 MHz FSB, 8GB memory) . They are connected via 1Gbps Ethernet links to a ZNYX ZX7000CA-X3 ATCA switch. That switch will then forward the stimulus load to the SUT connected for a particular test run. No other systems are connected to the switch.

5.7.2 The SUT Software

The SUT consists of software and hardware. While the hardware may vary, the software is almost the same (though even here various parameters can be tuned). However, since the hardware running the SUT depends on the experiment, it will be presented separately for each experiment.

The implementation for the IMS architecture used as SUT within the case study is OpenIMSCore [FOK06], the open-source implementation of Fraunhofer FOKUS [Fok08]. The System under Test is represented by the components of the Open IMS Core [VWM05]: P-CSCFs, I-CSCFs, S-CSCFs and the HSS. The HSS is a stateless Authentication, Authorisation, Accounting (AAA) server which uses MySQL [AB.08] for all of its state and data storage requirements. This function, called CHeSS, was developed by the same team at Fraunhofer FOKUS that developed the Open IMS Core. It uses the same CDiameterPeer routines as are used in the Open IMS Core,

and is built as a server using the same memory, locking, logging routines and basic structures as the SIP Express Router [Ipt07]. Its speed is mainly limited only by the database in use.

5.7.3 Visualisation Software

During the test execution, all benchmark-related information needed for assessing the SUT performance is collected and stored in the form of log files. Then, the log files are processed by the traffic analysis and choreography engine TraVis developed by Fraunhofer FOKUS. During this post-processing step, all the logged events are correlated and statistically analysed. As a result, the TraVis visualisation tool generates a benchmark report for the evaluation of SUT's behaviour under well-defined load conditions. This benchmark report contains various graphs to visualise collected metrics in different views: dependency on time, delays between events, stochastic distribution of events, etc. Also various statistics (max, min, average, variation, etc.) are reported. All graphs presented in the following sections are generated with the TraVis tool.

5.7.4 Experiment 1: A Complete Benchmark Execution Example

This section explains the way how the benchmark is executed against a concrete IMS core network. It also discusses how to interpret the various statistics and, the most important, how to determine the maximal load sustained by the SUT.

For this experiment, the SUT software is installed on a ATCA MPCBL0001 platform with 4x F29 2.00 GHz Dual Low Voltage Intel Xeon processors, 512KB L2 and 2Gb of memory. The server is running an RHEL AS4U3 operating system with a 2.6.9-34.ELsmp kernel.

Traffic Set

According to the test procedure, a complete benchmark test consists of several test steps with the purpose to determine the maximal sustained load of the SUT. The threshold for the termination of the test campaign is reached when the error rate percentage goes above a threshold. The values selected for the threshold of fails out of the generated load is 0.1%. The selected scenarios for the traffic set are presented in Table 5.1.

Table 5.1: Traffic Set Composition

Scenario Type	Label	Ratio
initial registration	s1.1	1%
re-registration	s1.2	1%
de-registration	s1.3	1%
successful voice-call without resource reservation	s2.1	12%
successful voice-call with resource reservation on originating side	s2.2	12%
successful voice-call with resource reservation on terminating side	s2.3	12%
successful voice-call with resource reservation on both sides	s2.4	12%
abandoned voice-call without resource reservation	s2.5	3%
abandoned voice-call with resource reservation on originating side	s2.6	3%
abandoned voice-call with resource reservation on terminating side	s2.7	3%
abandoned voice-call with resource reservation on both sides	s2.8	3%
rejected voice-call without resource reservation	s2.9	3%
rejected voice-call with resource reservation on originating side	s2.10	3%
rejected voice-call with resource reservation on terminating side	s2.11	3%
rejected voice-call with resource reservation on both sides	s2.12	3%
fail voice-call without resource reservation	s2.13	1%
successful page-mode messaging	s3.1	19%
failed page-mode messaging	s3.2	5%

Traffic Profile

The traffic profile which parameterises the test execution is given in Table 5.2. The benchmark starts with a stirring phase of the SUT for a period of 720 seconds followed by three load steps at 60, 70 and 80 SAPS. Each step is executed for 600 sec. The run uses 100000 subscribers and, out of these, 40% are registered at the beginning and used at the start of the tests. During the test, also other users from the rest up to 100000 will be registered but the number of active users will remain around almost the same since the registration and deregistration have the same frequency in the traffic set. This number may slightly vary due to registration/deregistration scenarios which make users active or inactive.

Table 5.2: Traffic Profile

Traffic Profile Parameter	Value
PX_StirTime	720 sec
PX_SimultaneousScenarios	2
PX_TotalProvisionedSubscribers	100000
PX_PercentRegisteredSubscribers	40%
PX_StepNumber	2
PX_StepTransientTime	120 sec
PX_StepTime	600 sec
PX_SApSIncreaseAmount	10 SAPS
PX_SystemLoad	60 SAPS
PX_PreambleLoad	42 SAPS

SAPS graph

In Figure 5.19 the main graph of the benchmark test is presented. The graph shows two shapes. The first one, above, presents the load applied to the SUT. It is intersected by several vertical lines which delimit the particular steps of the run. The test starts with a preamble step, where the users are registered to the IMS network. This is easy to recognise since the load is applied at a constant rate. The first vertical line marks the beginning of the stirring-time when the SUT is "warmed up" for the first load step. The stirring time is selected long enough to allow the SUT accommodate to the load, thus the test system avoids the effects caused by applying directly a high load. The stirring time load is actually split into three incremental steps so that the last step has the load close to the load of the first step. The end of the stirring time load is marked by the second vertical line. After the stirring time, the test system proceeds with the intended load test.

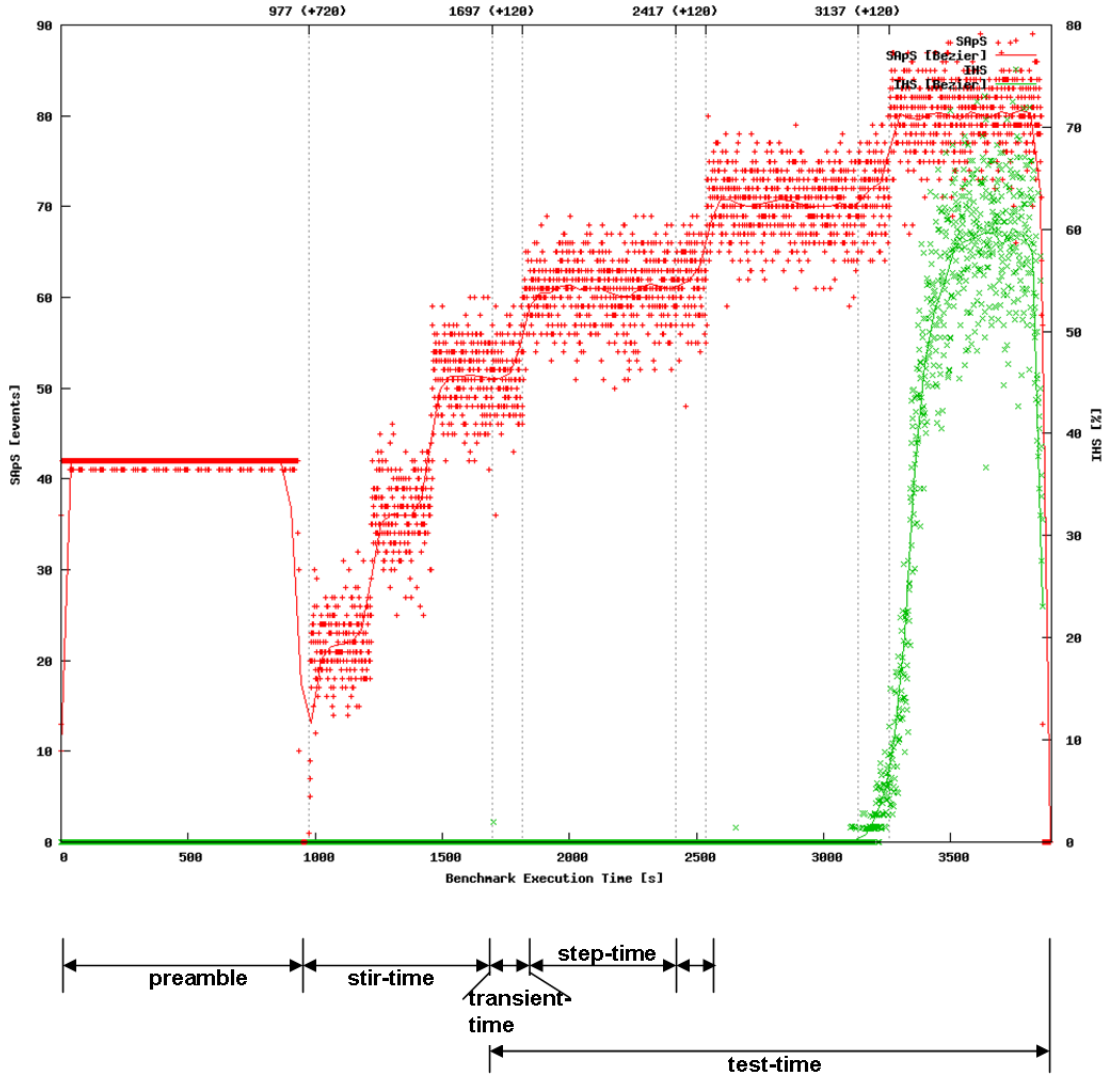

Figure 5.19: Steps to Determine the Load for which the SUT Starts Failing

At the beginning of each step, the load is applied for a short period of accommodation time called

transient time. During the transient time, the test system increases the load from one step to another, but does not compute the statistics. The reason for adding the transient time is to allow the test system to wait until the calls created in the previous step close and do not interfere anymore with the load which will be created in the next step. After the stirring time, the test executes three steps (at 60, 70 and 80), and at the start of each step, a transient time is marked by a vertical line.

The second shape displays the error rate which is computed as percent of fails out of the total created load. The error shape remains zero for a long period of time but starts increasing during the last step. In the middle of the last step it reaches 50-60%.

Obviously, the test system also detects that the SUT reached the limit of sustainable load by comparing in each step the IHS% average with the threshold. The IHS% maximal and average values are presented in Table 5.3.

Table 5.3: Inadequately Handled Scenarios Statistics

Step	Avg	Max
step1 (60SAPS)	0.00	0
step2 (70SAPS)	0.01	1.56
step3 (80SAPS)	49.28	73.08

The IHS% exceeds the threshold in the last step where it has an average of 49.28% fails. This means that almost half of the created calls are failing. This brings us to the conclusion that the DOC of the SUT has been exceeded. Based on the test procedure, the last load where the IHS% average is below the threshold is the actual DOC of the SUT. This happen in the example in the `step2` where the IHS% average is only 0.01.

IHS Graph

A more detailed view on the types of errors is provided in Figure 5.20 which displays the IHS% shapes per use case. This graph shows that the most occurred fails are from the session set-up/tear down use case, while the less fails are of type messaging.

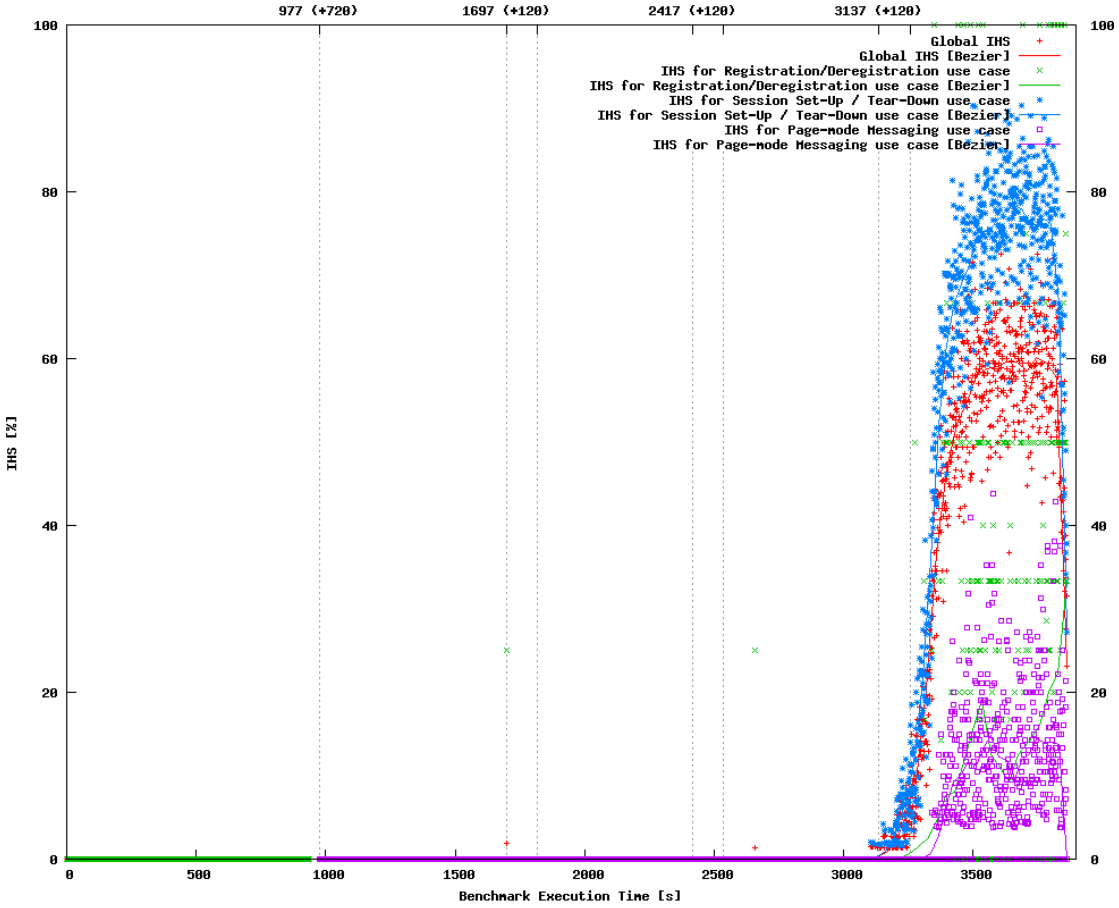

Figure 5.20: Visualisation of Fails per Use-Case

Error Causes per Scenario

The causes of errors can be investigated separately for each scenario. Figure 5.21 shows the errors statistics for the successful call with resource reservation scenario. During the first two steps, there is no error. But in the last step all error metrics increase. The most fails occurs at sending the first request (the INVITE) to the terminating party which means that the SUT is not able to process new calls after it reaches the overload capacity.

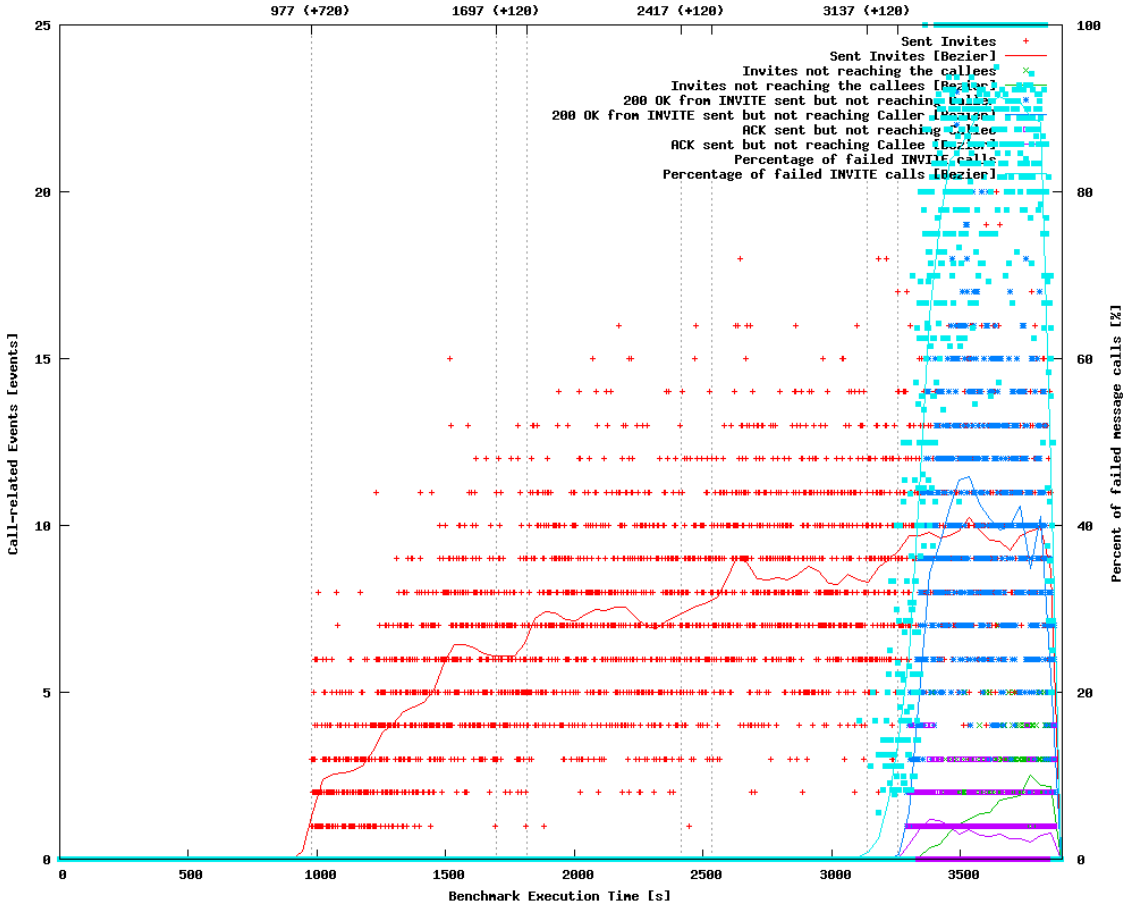

Figure 5.21: Error Statistics for the Successful Call with Resource Reservation Scenario

SIMS Graph

Figure 5.22 shows the SIMS (simultaneous scenarios) along the test execution. Depending on the scenario, the time a call remains open varies from a few seconds, e.g., registration, messaging, up to a few minutes, e.g., successful call. Therefore, it is expected that the value of SIMS metric indicate a large number of open calls compared to the number of SAPS.

In a normal behaviour, this metric should stabilise around an average value if the SAPS is maintained constant. In this figure, after a normal shape along `step1` and `step2`, the SIMS values grow dramatically in the last step which actually means that the SUT receives too many calls and it can not finish the existent ones. The SIMS shape does not seem to stabilise, but on the contrary it rises continuously until the end of the test. This is another clear clue that the SUT reaches its DOC in the last step.

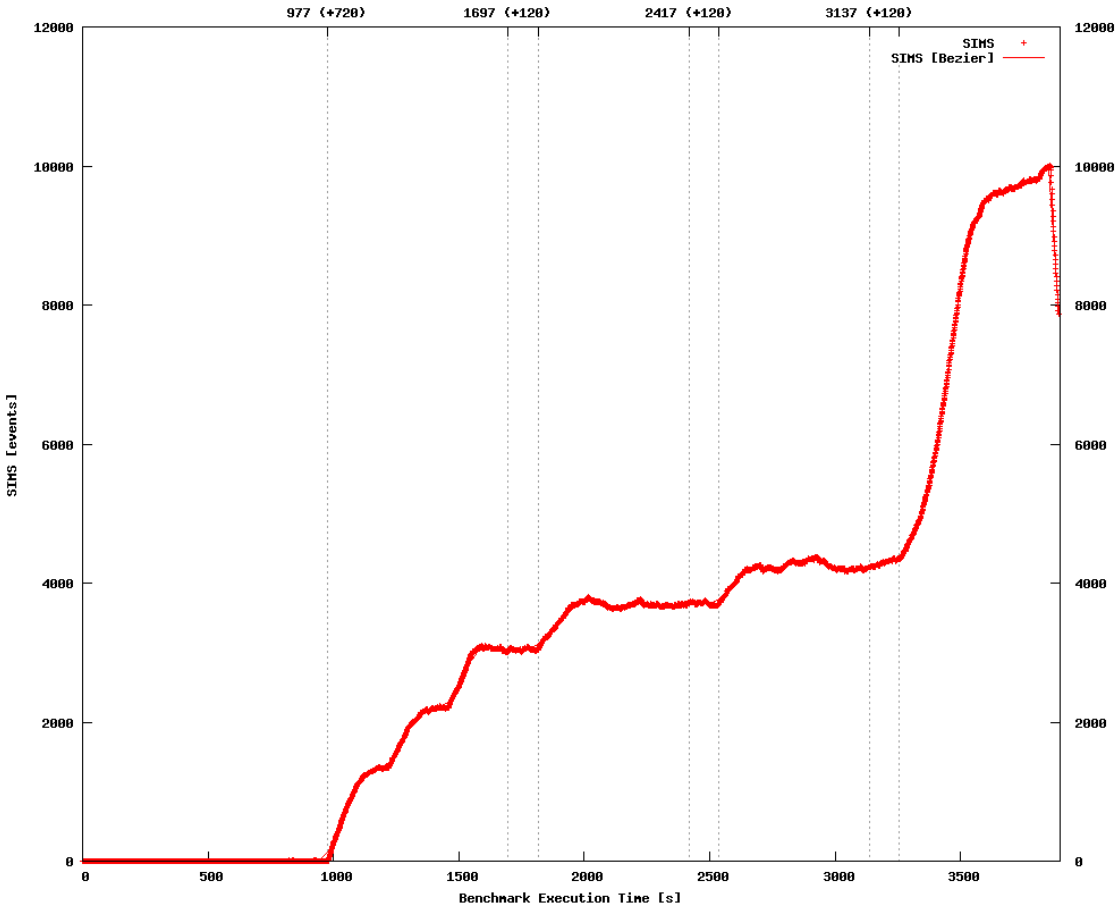

Figure 5.22: Visualisation of Simultaneous Scenarios

RETR Graph

In a further graph (see Figure 5.23, the RETR (retransmissions) metric is visualised for each use case. The RETR metric is shown in parallel with the SAPS metric so that the load intensity can be correlated with the RETR values. In a normal SUT behaviour, the number of retransmissions should be very low (ideally no retransmissions). During `step1` and `step2` the RETR is close to zero. In the last step, the test system has to retransmit many messages since they are not answered in time by the SUT.

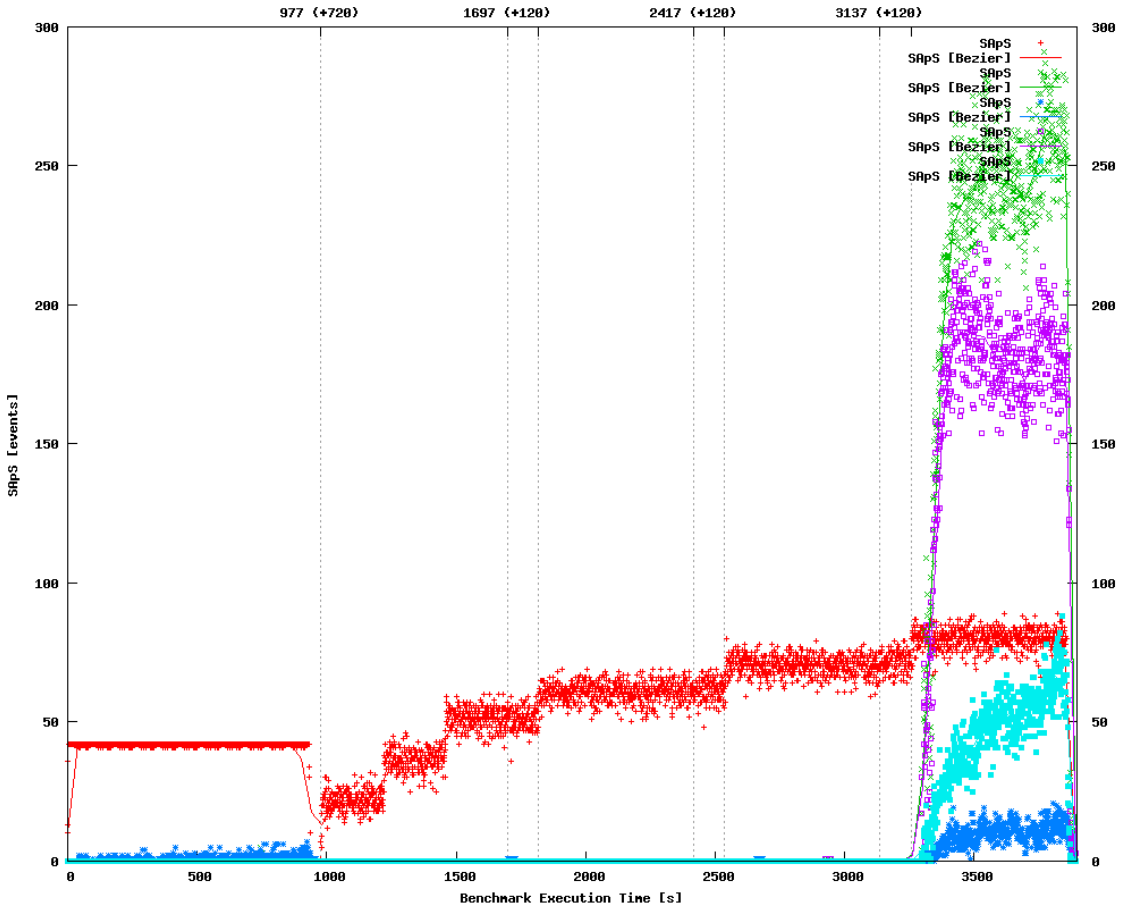

Figure 5.23: Visualisation of Message Retransmissions

CPU Graph

No doubt, the best view of what happens around DOC is given by the CPU monitoring graphs. Figure 5.24 shows the idle, system and user times of the CPU. It is obvious from this graph that the DOC is reached when the SUT almost runs out of resources. For example, the idle shape decreases from 50% during the first step to 10% in the last step. Mirrored, the user time increased from about 40% up to 80% which indicates the high demand of CPU on the SUT side.

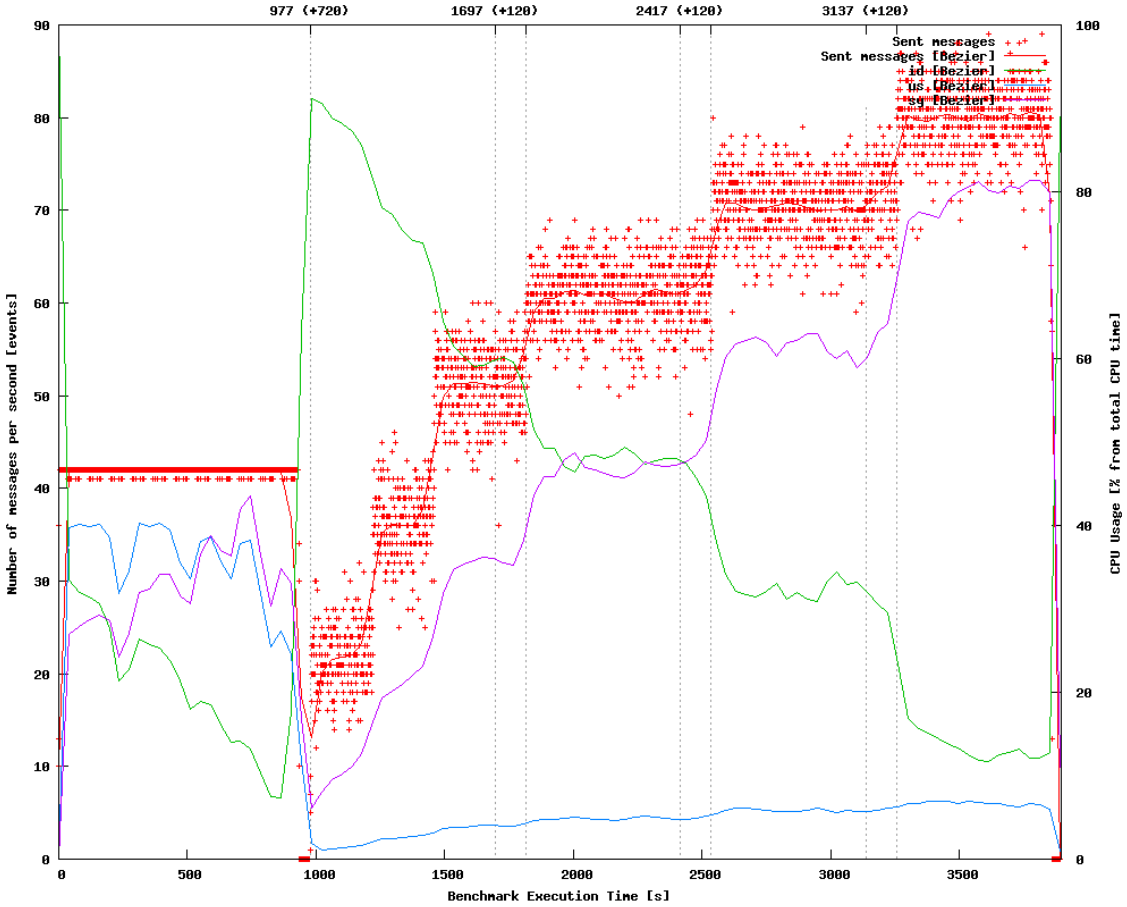

Figure 5.24: CPU Consumption

MEM Graph

An abnormal behaviour of the SUT around DOC is also noticed from the memory consumption graph presented in Figure 5.25. While during the first two steps, the free memory decreases very little, it has a huge fall in the last step indicating that the SUT needs much more memory.

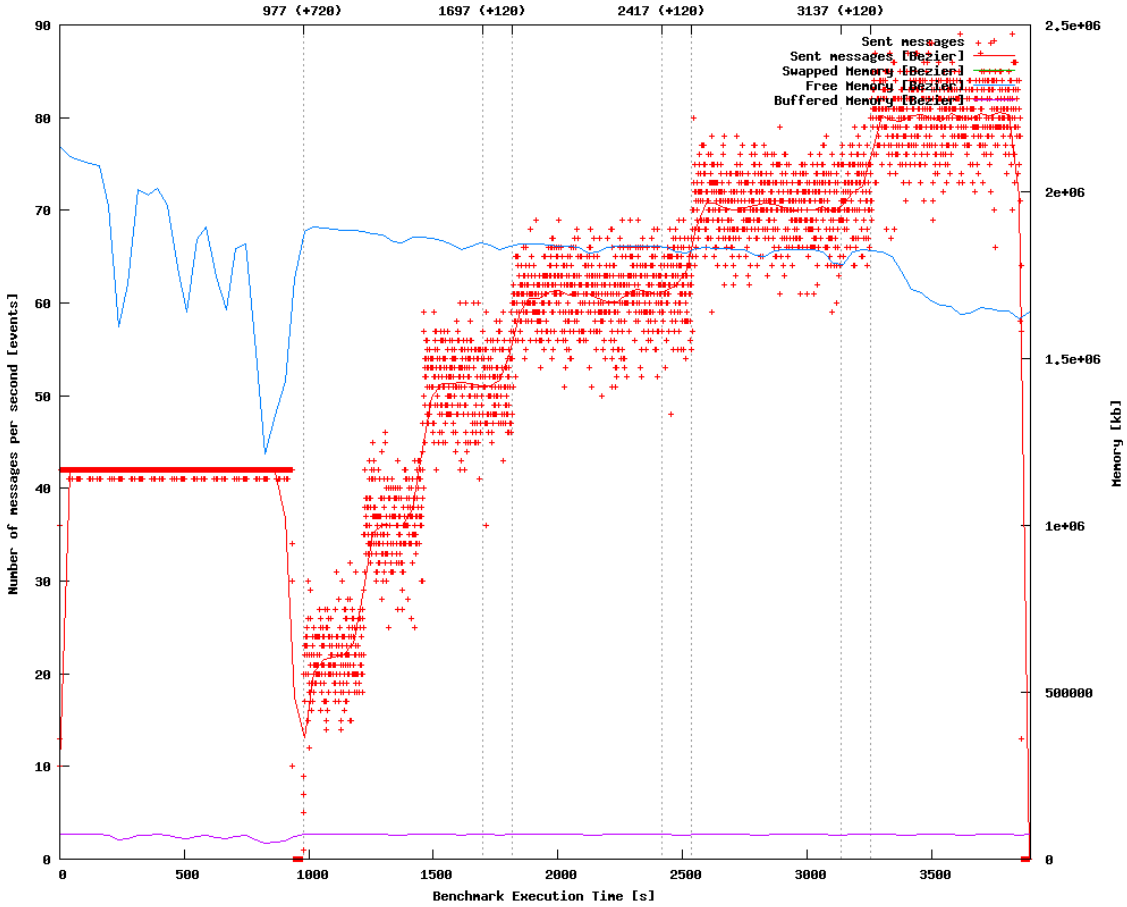

Figure 5.25: Memory Consumption

Test System CPU Monitoring

To validate the results, it is necessary to prove that the test system was not overloaded during the test execution. This can be inspected by looking to the CPU consumption of each test system server. One of these graphs is presented in Figure 5.26. Along the test, the test system barely used the CPU resource, almost 90% being idle, except the last step when the test system had to deal with the retransmission (which require more CPU).

Figure 5.26: Test System Server CPU Consumption

Long Runs to Prove that the SUT Runs Stable at the DOC Load

Once the DOC has been found, it is important the check that the SUT behaves stable also for longer runs. A long run, for at least half an hour, discovers situations when the SUT can hold the DOC load only for short runs but not for longer runs. Ideally, all steps in the test procedure should be long runs but to avoid very long test executions, the shorter runs are used to find the DOC and, at the end, several confirmation runs prove the DOC value. Figure 5.27 shows that the SUT remains stable also for a period of 30 minutes.

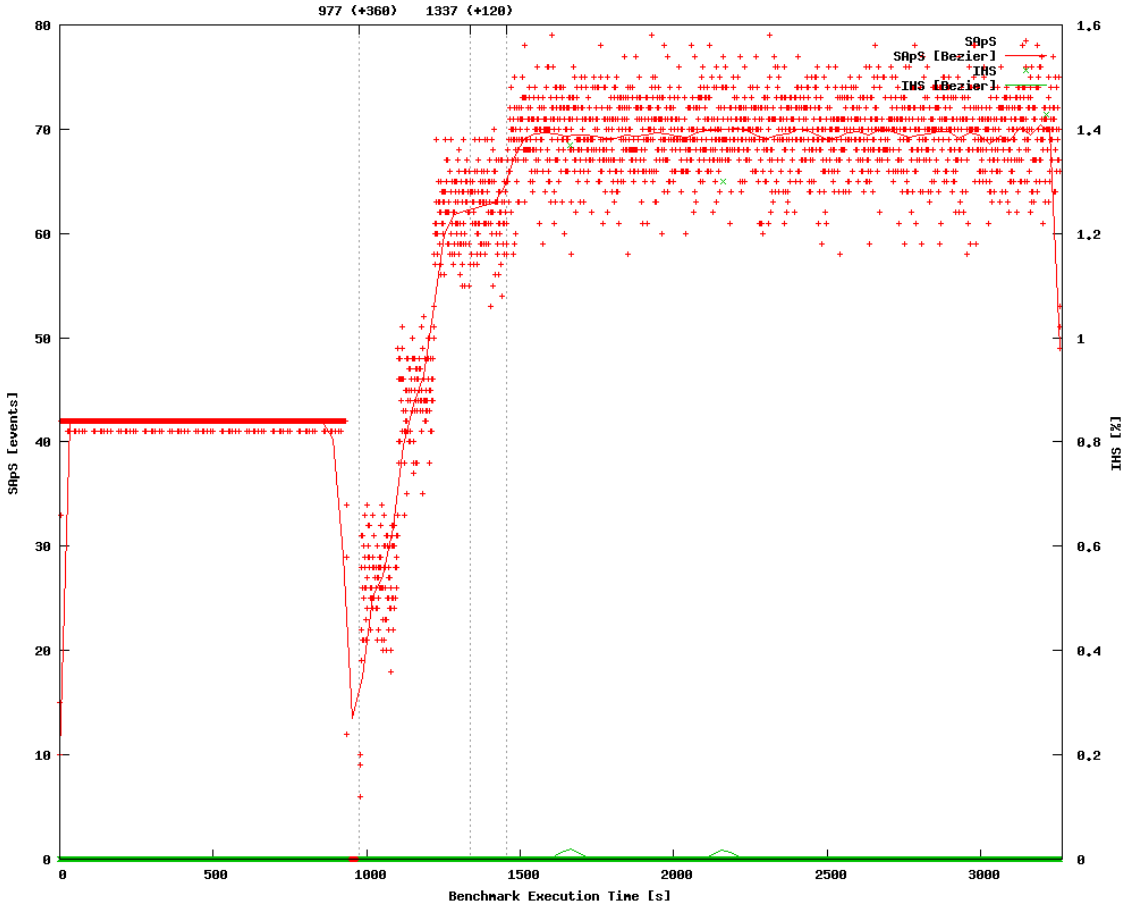

Figure 5.27: Long Run to Check the Stability of the DOC

5.7.5 Experiment 2: Hardware Configurations Comparison

The benchmark may serve as a comparison tool between different hardware and software configurations. The comparisons based on IMS benchmark can be realised in different ways:

- A - comparison between different software configuration with same hardware. This approach is useful for the software developers as a debugging method. Along the project one may track the performance improvements. Additionally, the approach is useful also for comparing different tuning of the SUT software on a given platform.
- B - comparison between different hardware configurations but same software configuration. This approach can be used by hardware vendors in order to compare their hardware on a common workload basis.
- C - comparison between two different hardware configurations and different software configuration. This case reflects the situation when the software is tuned for a particular hardware configuration in order to take advantage of the underlying hardware resources.

Table 5.4: Benchmark Comparison of Different Software Configurations

Hardware Configuration	DOC
With activation of monitoring modules	60
Without monitoring modules	180

Table 5.4 presents the performance results of two experiments with different software configurations running on the same hardware. For both experiments, the test system is running the same traffic set and traffic profile. The SUT is first configured with some monitoring capabilities activated, while in the second experiment, these features are deactivated. The performance number in the second experiment is three times higher than in the first experiment which reveals that the features activated in the first experiment heavily impact the performance. This is an essential performance debugging information which helps the SUT engineers localise and improve performance bottlenecks.

Table 5.5: Benchmark Comparison of Different Hardware Configurations

Hardware Configuration	DOC
mem=4GB cpu=4x2.00GHz cache=512KB L2	180
mem=8GB cpu=4x2.00GHz cache=2MB L2	270
mem=8GB cpu=4x2.66GHz cache=4MB L2	450

Table 5.5 presents a comparison of the DOC numbers for different servers. The SUT software has been installed with the same configuration on all servers. The DOC is obviously higher for the last two servers which have more memory. However, the cache seems to have the biggest impact since the last board (with the highest DOC) has similar configuration as the second server but more cache.

Due to lack of time, experiments of type C have been left out. These kinds of experiments are the ultimate tests for comparing the optimal configurations of hardware and software. They are

utilised for managerial decisions when selecting the most convenient configuration in terms of costs, performance and number of supported users.

5.7.6 Traffic Set Composition Experiments

The DOC is measured according to a traffic set selection. Therefore, an IMS network may be able to sustain a certain load for a traffic-set1 but not for a traffic-set2. The concept of traffic selection allows the testers identify a minimal capacity of the network for which all traffic sets of interest can be successfully sustained.

5.7.6.1 Experiment 3: Successful Scenarios vs. Mix of Successful, Abandoned, Rejected, Failed Scenarios

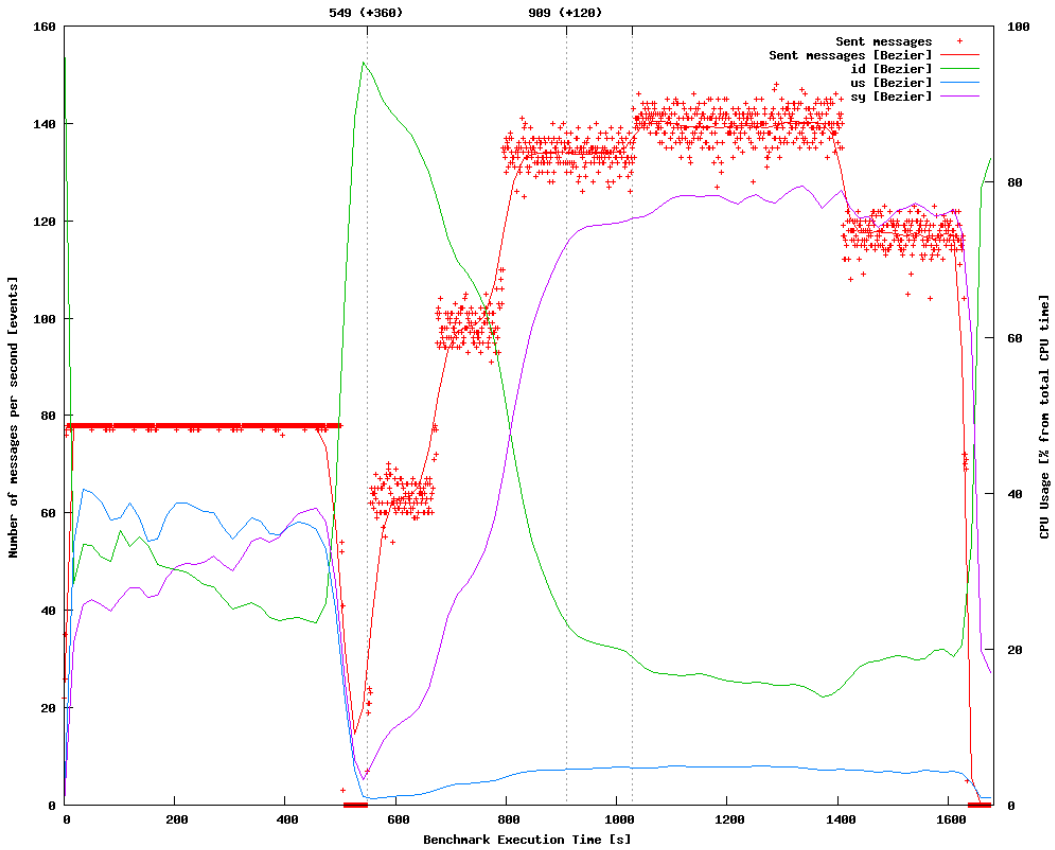

Figure 5.28: CPU Consumption Using Only Successful Scenarios

This experiment shows the influence of mixing successful scenarios with abandoned, rejected or failed scenarios. Figure 5.28 shows the CPU consumption for a traffic set which consists of only successful scenarios. Figure 5.29 shows the CPU consumption for a traffic set which consists of scenarios from all categories. In the second run the SUT uses less CPU since the fails are easier to be handled than the successful calls. This happens because the successful calls include also the talking time period which requires the SUT to keep the state of the call in memory for that period of time. This contrasts with the abandoned or rejected scenarios which do not have to stay active

as long as the successful ones. This result can help service providers estimate correctly the capacity of their systems. However, this estimations rely on statistics which indicate the proportions of abandoned or rejected calls out of the total number of calls.

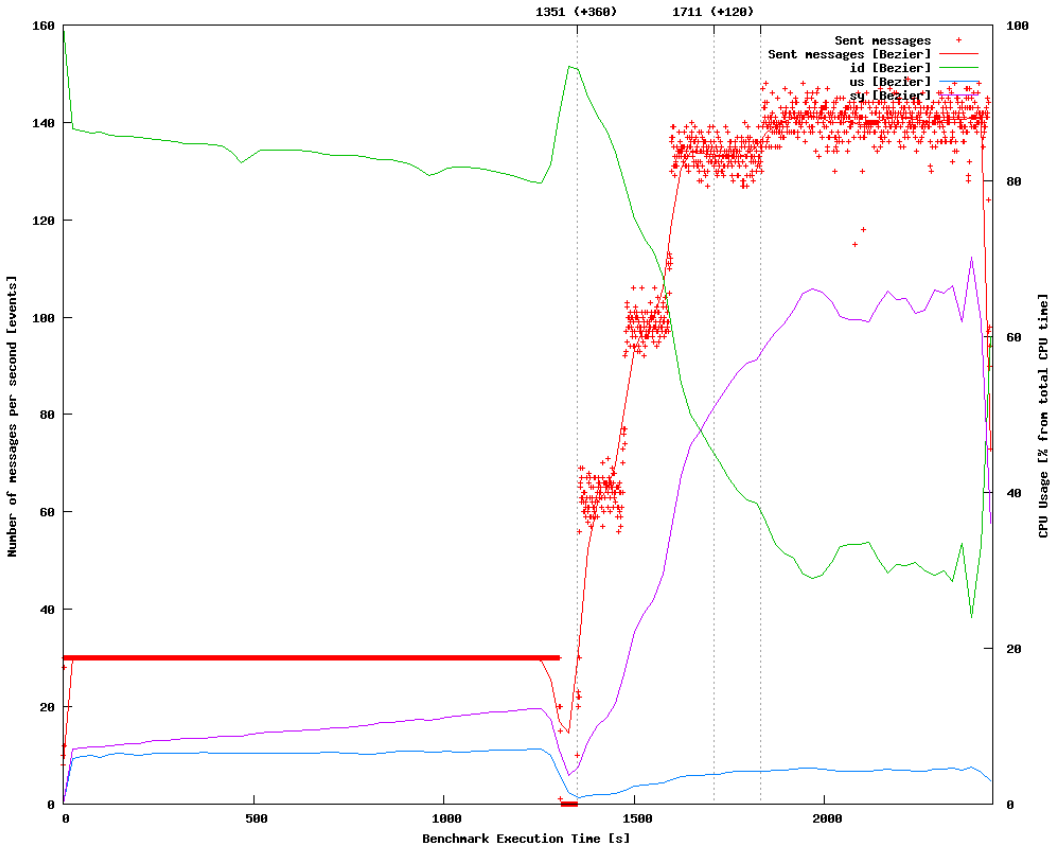

Figure 5.29: CPU Consumption Using a Mix of Successful, Abandoned, Rejected and Failed Scenarios

5.7.6.2 Experiment 4: Traffic Set Combination

To obtain valuable results, the traffic set must be carefully selected. It is obvious that for different traffic sets, the SUT has different capacities. This result can be observed also from the following experiment. Two different traffic sets are selected. The first one is the traffic set used also in Experiment1 (see Section 5.7.4). The second traffic set contains the same scenarios selection but with slightly different proportions (moved 2% from successful scenarios to abandoned scenarios).

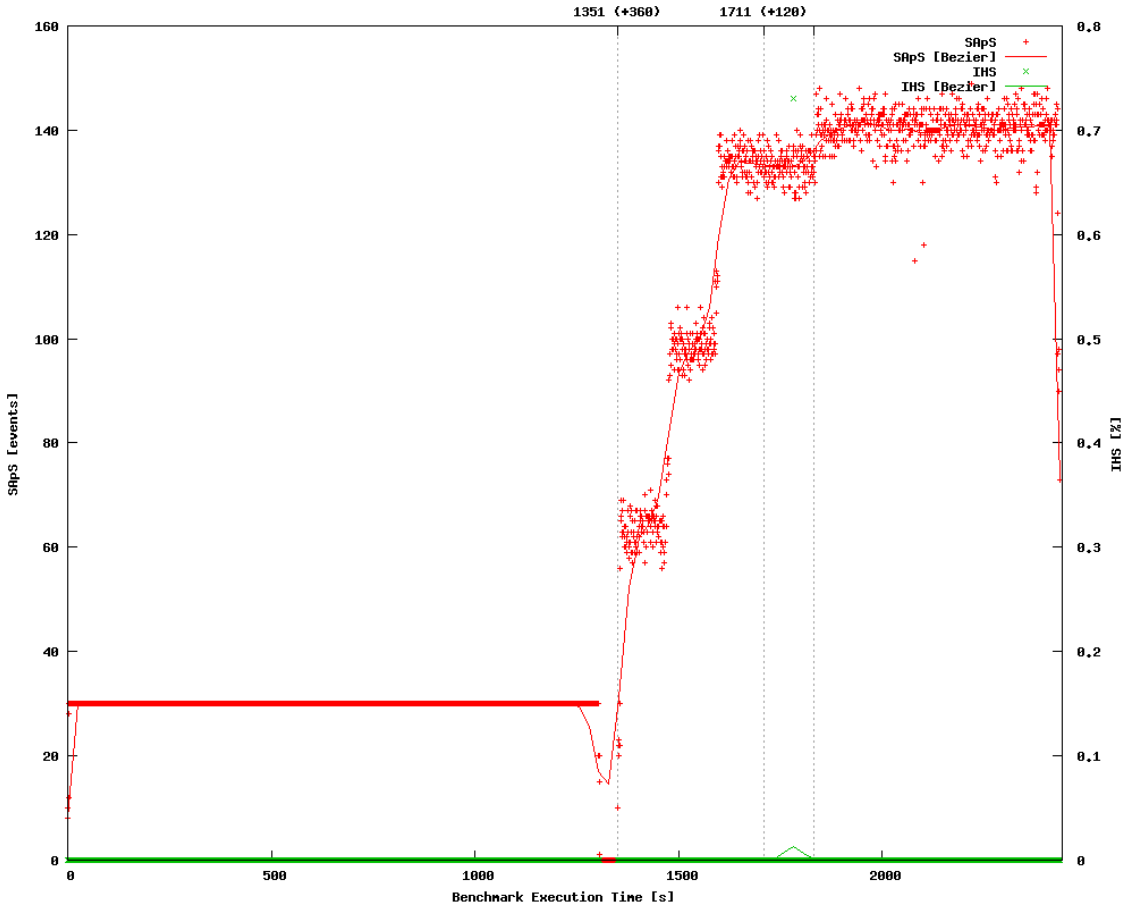

Figure 5.30: First Traffic Set Including Only Successful Scenarios

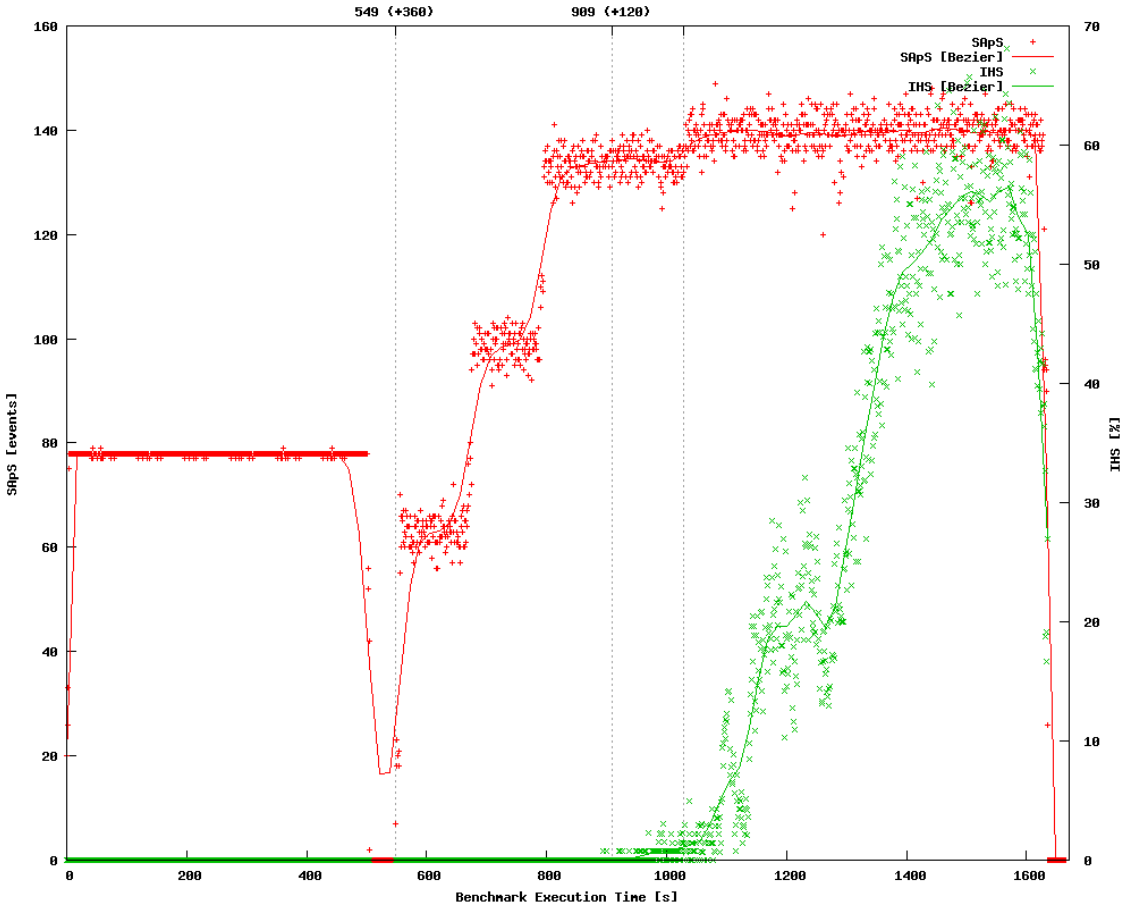

Figure 5.31: Second Traffic Set Including Abandoned, Rejected and Fail Scenarios

5.7.7 Benchmark Parameters which Impact the SUT Performance

The performance results are influenced not only by the traffic set combination but also by the traffic profile parameters. The following experiments show how the modification of these parameters acts upon the resource consumption on the SUT side.

5.7.7.1 Experiment 5: User Number

The number of active users influences both the memory and CPU consumption of the SUT. Memory is allocated to store the status information of each user. Therefore, the more users are registered, the more memory is allocated. The CPU is needed to handle the transactions and subscriptions.

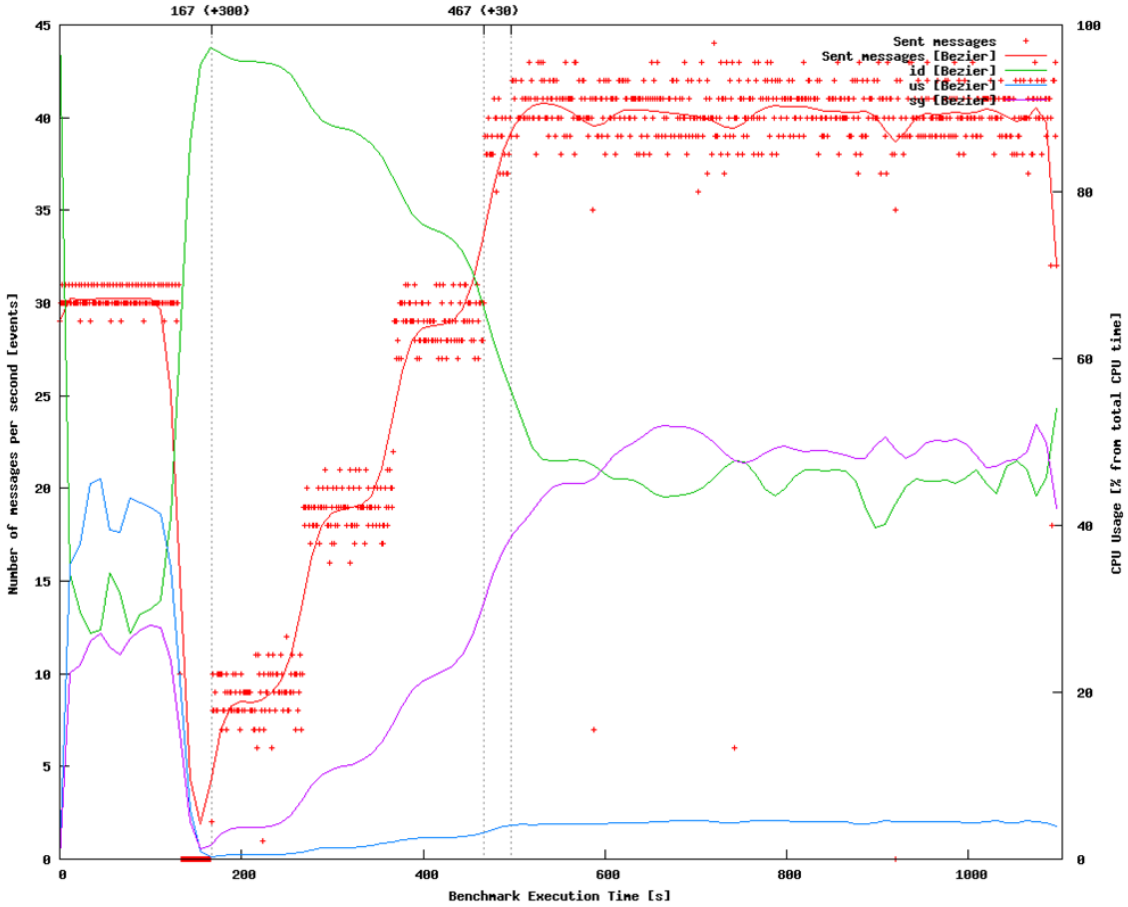

Figure 5.32: CPU Consumption for 5000 Users

Figures 5.32 and 5.33 show the CPU consumption for two tests with the same workload but using different number of users. While in the first test the free CPU is about 40%, in the second test the free CPU reaches 0%. It might look strange that the second experiment uses more CPU for the same number of transactions, but the difference appears due to internal users state management (the SUT reserves more memory in the second experiment).

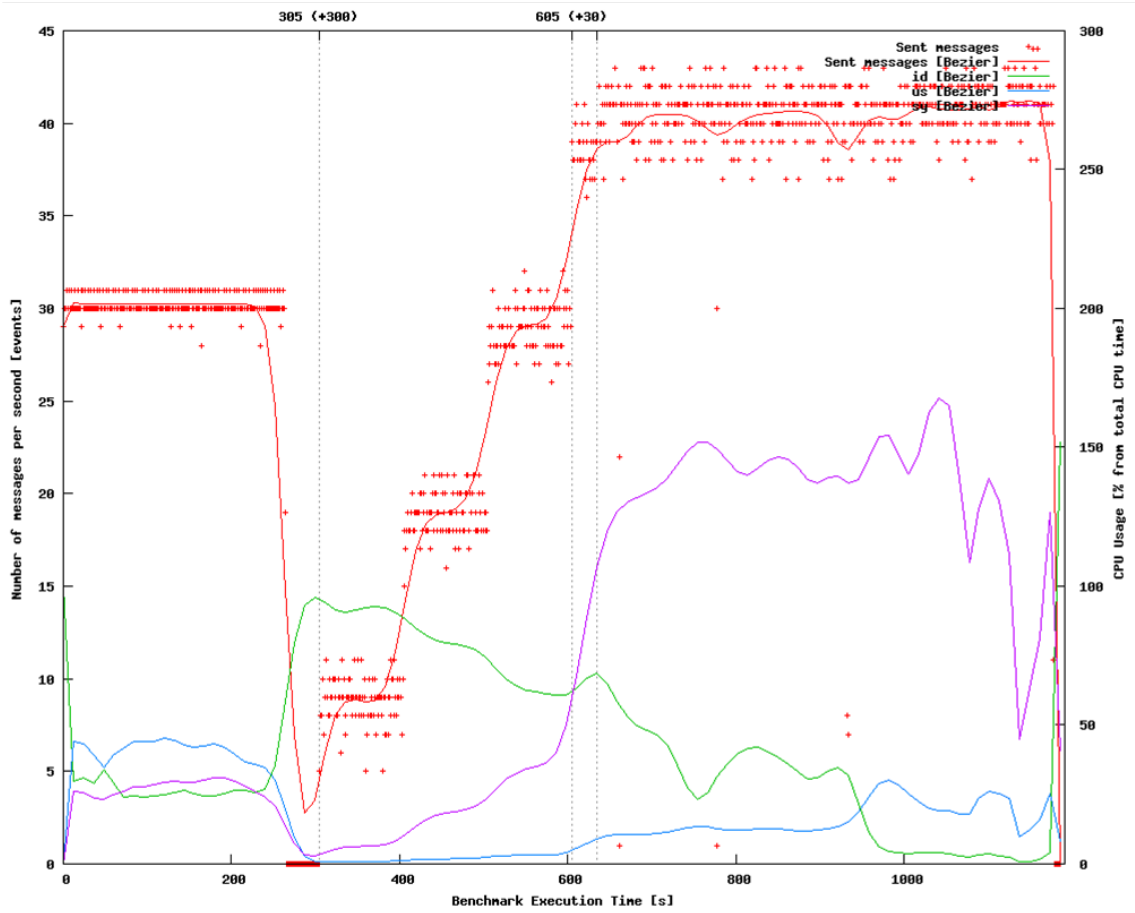

Figure 5.33: CPU Consumption for 10000 Users

5.7.7.2 Experiment 6: Execution Time

For accurate results, the execution times should be long enough so that a large number of scenarios are executed. Figure 5.34 shows a test with two steps at 320 and 330 SAPS, executed for 10 minutes each. During the first step, the SUT holds the load with very few fails. In the second step, the SUT becomes overloaded and ends up failing all calls. According to this test, it seems that the DOC is 320 SAPS. Running another test for the same traffic set but for 30 minutes, reveals that the SUT reaches the overload limit after 15 minutes. This implies that the DOC is below 320 SAPS.

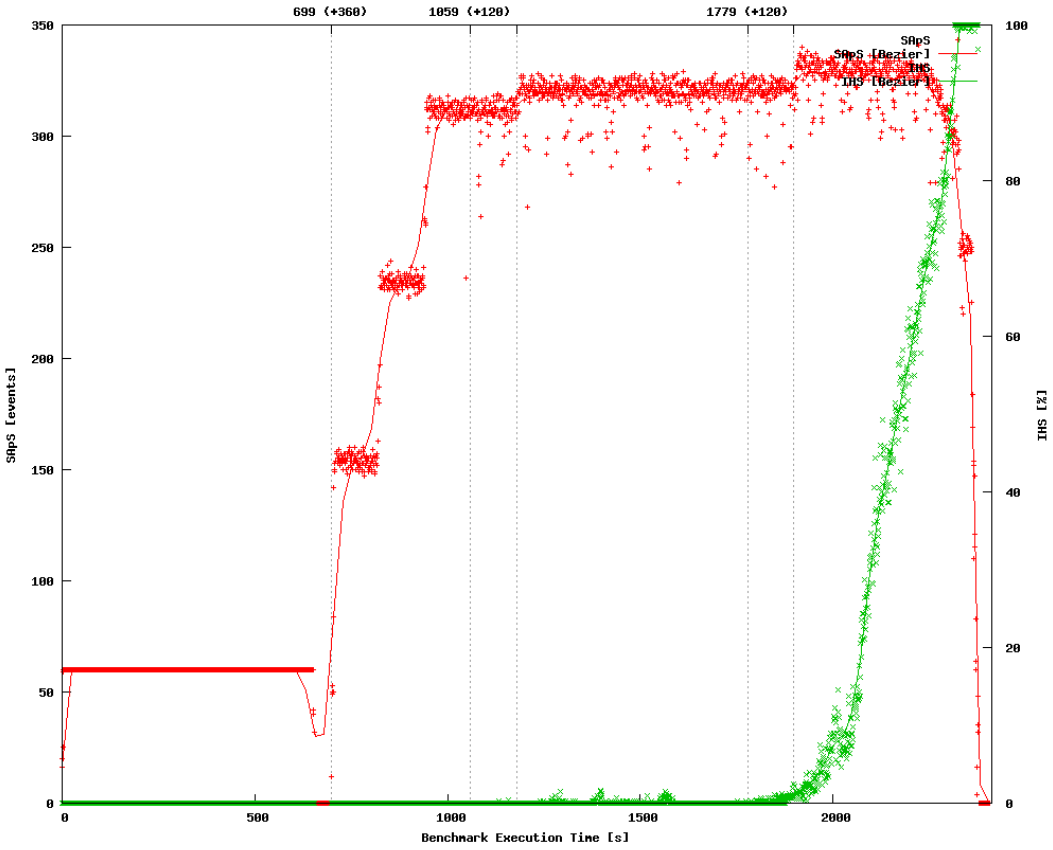

Figure 5.34: Test Run with 320 SAPS for 10 Minutes

Table 5.6: Execution Time Durations Corresponding to 1.000.000 SAPS

SAPS/sec	Duration
100	2.7 hours
200	1.3 hours
300	55 min
400	41 min

This experiment brings to the conclusion that the SUT may sustain higher loads but only for short period of times. The duration is actually influenced by the number of total calls created along the test. For different loads but same duration, the TS create different numbers of calls. An ade-

quate value for the number of calls is one million of calls. This number permits the computations of statistics with very small error rates but also should be huge enough to do not allow the SUT "survive" as it does for short runs. Table 5.6 shows the optimal durations for one million of calls computed for different loads. For 100 SAPS the test should be executed for 2.7 hours while for 400 SAPS only 41 minutes should be enough.

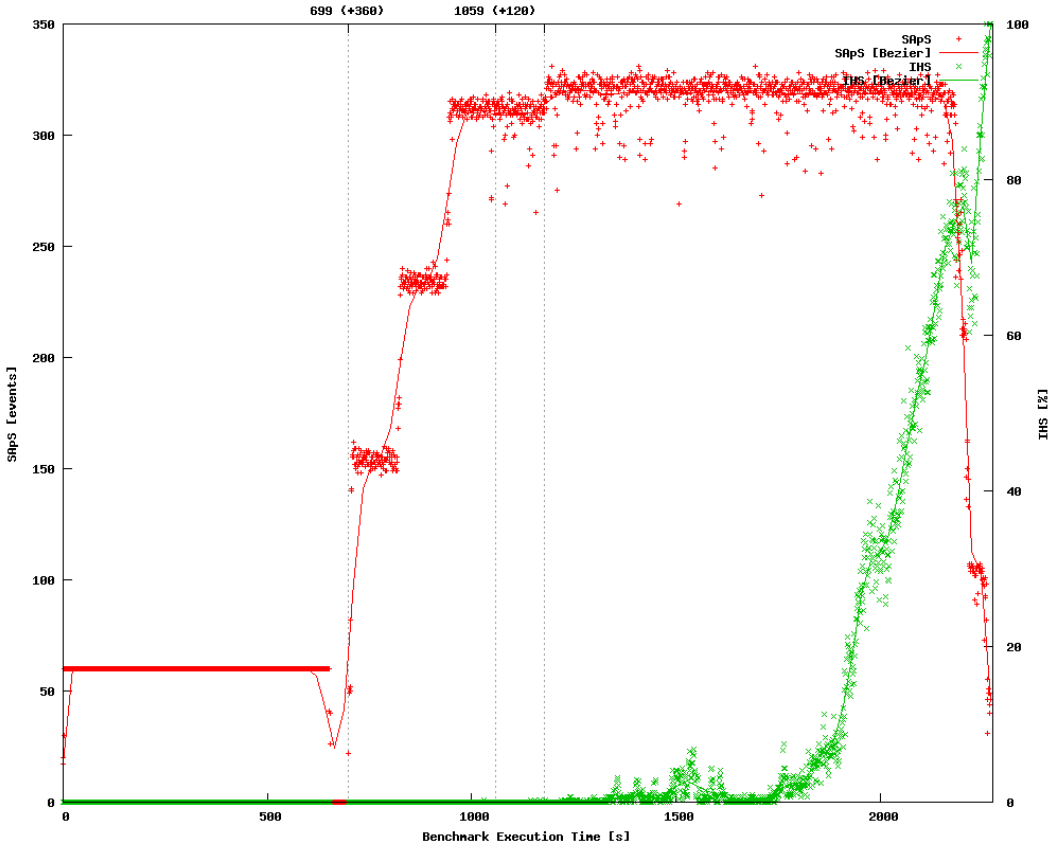

Figure 5.35: Test Run with 320 SAPS for 30 Minutes

5.7.7.3 Experiment 7: Stirring Time

The stirring time is a parameter representing a period of time in the preamble of a benchmark test in which a system load is run in order to allow initial transient conditions attenuate to an insignificant level. Figure 5.36 shows the effect of setting the stir time to null. The SUT fails some of the initial calls right after the beginning of the test, thus affecting the performance statistics.

A long enough stir time, e.g., 10 minutes, should avoid this effect, as it is observed in the Experiment 1 (see Section 5.7.4) when this effect does not occur.

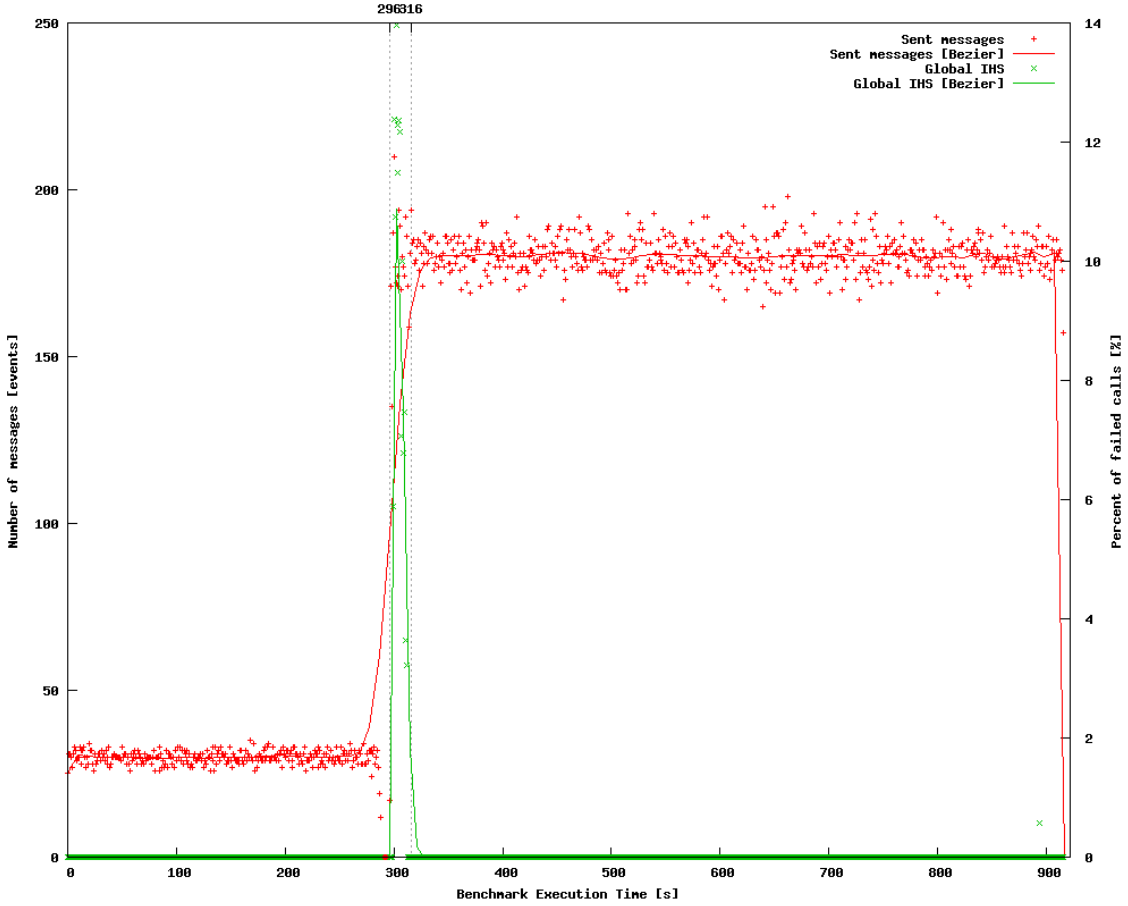

Figure 5.36: Test Run without Stir Time

5.7.7.4 Experiment 8: Transient Time

At increasing the load to a new step, there is a period of time when the SUT is exposed to the new load but when it also have to handle the existent calls. Since the existent calls will have to terminate during the next step, there is the possibility that the SUT is overloaded by the new load and the previous call terminations fail. Due to this phenomenon, the previous step may exceed the fails threshold only because the termination of the last created calls happens during the second step.

Figure 5.37 provides an example for how the last created calls fail only because the SUT reaches the overload capacity in the last step. To avoid this effect, the transient time has been introduced. A safe value is usually a value long enough to ensure that all calls from the previous step are terminated.

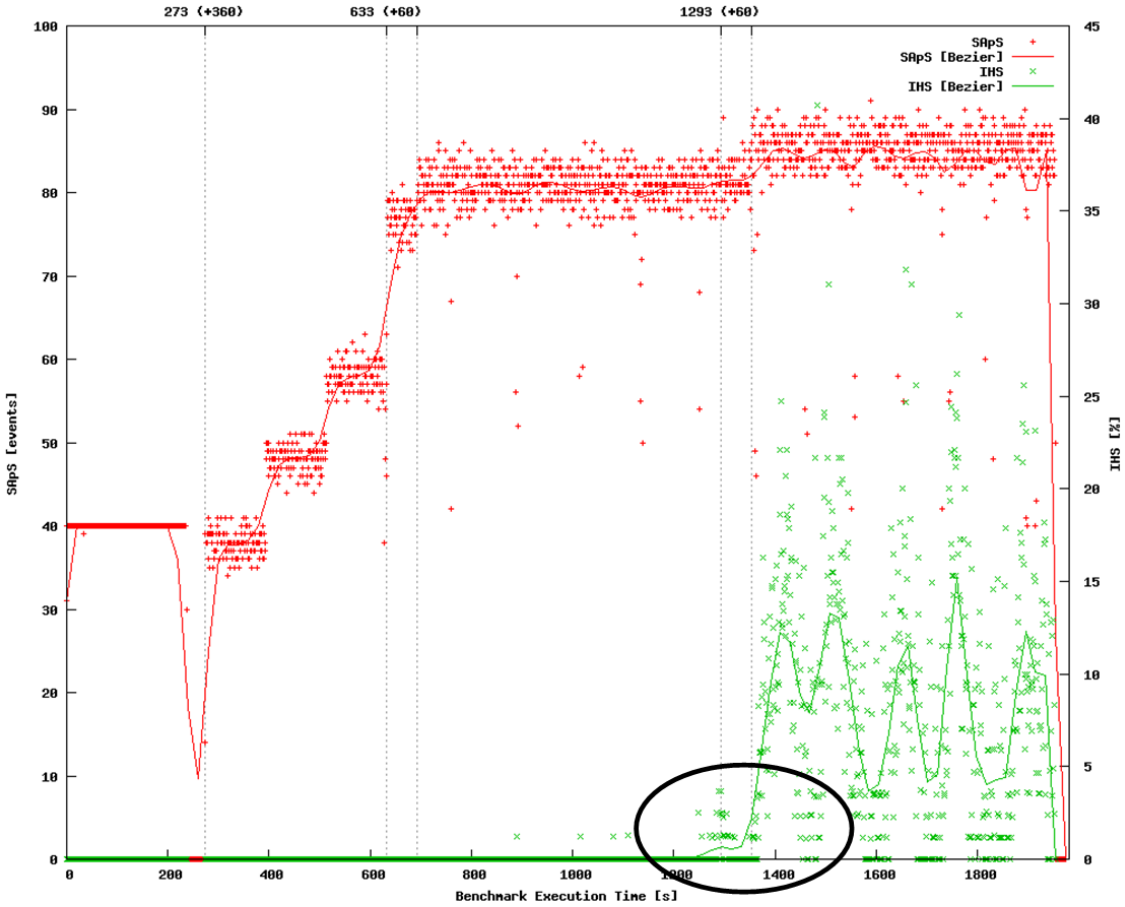

Figure 5.37: Transient Time

5.8 Summary

In this chapter the IMS benchmark case study has been presented. It aims at developing a performance benchmark for comparing IMS deployment configurations. The IMS subsystem architecture has been introduced and the performance testing elements used to design the benchmark have been highlighted. A number of experiments have been conducted to determine the capacities and compare the performances of IMS deployments on different hardware configurations.

A large part of the chapter is dedicated to exploring the parameters which influence the performance results. It is revealed that many of these parameters have a great impact on the SUT performance. Due to the need for realistic experimental conditions, the values of these parameters are analysed.

Chapter 6

Conclusions and Outlook

Let the future tell the truth, and evaluate each one according to his work and accomplishments.
The present is theirs; the future, for which I have really worked, is mine.
– Nikola Tesla

6.1 Summary of the Thesis

The performance testing of continuously evolving services is becoming a real challenge due to the estimated increase of the number of subscribers and services demand. More efficient and more powerful testing solutions are needed. This ability highly depends on the workload design and on the efficient use of hardware resources for test execution.

This thesis presented a methodology for performance testing of multi-service systems. The topic embraces the challenges of the nowadays telecommunication technologies which are characterised by a huge variety of services and interactions. The main output is the method to create workloads for performance testing of such systems. The approach takes into account many factors: test scenarios, test procedures, metrics and performance test reporting. The design method ensures that the workloads are realistic and simulate the conditions expected to occur in the real-life usage. The main performance indicator is the DOC which reports the largest load an SUT can sustain and it serves as comparison criterion among different SUT implementations, SUT hardware configurations etc.

The thesis approaches also the test harness topic in the performance testing context. It establishes the requirements related to test harness for the characteristics specific to performance testing (including test distribution). The TTCN-3 language has been selected to allow a more concrete realisation of the presented concepts. The arguments for the selection of this technology are also provided. The language is then used to show how various performance testing concepts introduced in the testing methodology such as event handling, traffic set, data repositories, etc., can be designed. With respect to test execution, an extension of the ETSI standardised architecture for test execution is discussed. Various aspects such as test management, test execution and distribution are discussed in detail.

Within the case study, a performance test suite capable of evaluating the performance of IMS networks for realistic workloads has been designed and developed. The results have been published as IMS Benchmark TISPAN technical specification [TIS07]. The available implementation provides

23 IMS scenarios with varying traffic sets and increasing traffic loads and it is used to differentiate the performance of different IMS core network configurations. The software is now available at Fraunhofer FOKUS [Fok08] and it is marketed by the Testing Technologies company [TTe08].

The use of an abstract language such as TTCN-3 to specify the performance tests was worth for a number of reasons. From a technical perspective, the abstract level helped to better organise the sequence of interactions between users and SUT. The message receiving mechanism simplified the creation of further messages directly from the received message by using a very small amount of code. In addition, most of the processing tasks required small amounts of TTCN-3 code, thus improving the readability of the specification. But, along with the ease-to-use also comes more overhead and resource load on the test nodes and it becomes even more important to look at the performance issues. This motivated the work toward implementation patterns.

The case study helped to experiment the strategies to design and implement performance test systems with TTCN-3 technology but also revealed the technical issues of the whole solution. Compared to other technologies, e.g., SIPp [JG06], the TTCN-3 and TTmex [TTm08] combination faces more performance issues but gains from simplicity, extendability and state control in the test specification.

The workload distribution over multiple computation resources is the obvious method to scale the performance of a test system, in order to supply a large amount of virtual users. In this respect, it has to be considered that the hardware involved in performance testing is expensive and many performance test beds are constructed on heterogeneous hardware. This aspect has also been addressed within this thesis by using intelligent distribution algorithms of the workload thus scaling the test system performance needs.

6.2 Evaluation

The target of this work was to realise rather a generic framework for performance testing than a single purpose implementation, making it easier to extend it for new SUTs, new types of user-SUT interactions, new protocols, etc. Therefore, the design process described in Chapter 3 can be applied directly to a new SUT.

This thesis can be evaluated by analysing the value and novelty of contributions it makes:

- *performance test design process* - Different to previous approaches, the test design process proposed in this thesis takes into account the representativeness of the selected use cases by emphasising new concepts such as design objectives, traffic set, traffic-time profile. Additionally, it is proposed to follow up a test procedure which unquestionably leads to the determination of a *single, representative and comparable performance indicator* - **a simple number**. This number, which is the DOC, is the introduced criterion to differentiate the performance of various systems irrespective of the used test system.
- *implementation patterns and architecture* - No other work before has collected and combined in such a comprehensive manner the test system implementation possibilities regarding: *abstraction layer*, e.g., test specification language, *underlying platform*, e.g., Java, C, and *operating system layer*, e.g., Unix, Windows. This has been achieved by identifying a set of patterns to be instantiated during the test implementation process. The usability of these patterns has been analysed in a comparative manner with respect to effort and resource consumption.

- *TTCN-3 based specification* - By selecting the TTCN-3 language as implementation language, this thesis also contributes significantly to the evolution of this language toward non-functional testing. Since its standardisation, TTCN-3 has been used especially for conformance testing. This work shares the knowledge of how to realise performance tests. It may happen that other solution such as Spirent Tester, IxVoice or SIPp overtake the TTCN-3 based solution in terms of performance, but with TTCN-3 one gains in terms of a) use of a standardised test notation, b) use of the same notation for all types of tests, c) specify the tests at an abstract level and d) use a generic and flexible solution.
- *IMS test solution* - As far as the case study is concerned, another value of the thesis resides in the contribution to the core parts of the TISPAN technical specification for IMS Benchmarking.

6.3 Outlook and Possible Extensions

The presented methodology targets the structure of performance tests, therefore it can be applied to many types of systems. Its applicability to further domains is foreseen: Web services, banking systems, automotive etc. However, the idea of applying the methodology into other areas is already in progress [DES08]. In this respect, a significant extension of this work would be the formalisation of the performance test design elements in a modelling language such as Unified Modeling Language (UML). This will open the possibility to apply model driven test generation techniques [Dai06] to the performance test design and, thus, automatically generate performance tests out of UML models. At the model level many aspects of performance test design can be treated a way easier by using class and interaction diagrams.

One of the most common questions in performance test system engineering is "how to maximise the test system throughput?". Typical strategies to improve the speed of a system include [Ins06]: efficient resource management by smart implementation, techniques to take advantage of the latest processor technology and/or design of system architectures that support parallel test and resource sharing. The approach presented in this thesis investigates the performance engineering by looking into various patterns to implement a performance test framework. The list of patterns can still be extended if further hardware or OS artefacts such as Hyper-Threaded CPUs, multi CPU architectures are considered. Additionally, platform, e.g., J2EE, specific characteristics should be taken into consideration.

Another way to improve the performance of the test system is to efficiently balance the load over the computation resources. The problem of efficient allocation of hardware resources or resource scheduling is encountered [SHK95] very often. The complexity of the tested systems is increasing rapidly as well as the demand for higher quality and more features. The performance testing tools for typical client-server architectures require the simulation of thousands of users, e.g., IMS, UMTS; this requirement rises the problem of how to design a proper system to sustain the effort of thousands of users by using a cost acceptable hardware. To achieve that, the load balancing mechanisms play a major role in the execution dynamic and comprises activities like distribution of users, distribution of the computation effort, balancing of CPU consumption among the parallel processes. In this respect, the framework can be extended with various algorithms for dynamic distribution and load balancing.

Glossary

Benchmarking
Benchmarking is a performance test used to compare the performance of a hardware and/or software system. In general, benchmarking is similar to a load test which has a well defined execution procedure which ensures consistently reproducible results.

Design Objective
A design objective is a performance requirement defined for a scenario and expressed as delay or as failure condition. Design objectives are specified as threshold values for the evaluated performance requirements.

Load testing
Load testing is a type of performance test which simulates various loads and activities that a system is expected to encounter during production time.

Metric
A metric is a computation formula based on performance measurements of an SuT reported in a performance test report. This term is used interchangeably with performance metric term.

Multi-service system
A multi-service system offers a number of services which can be accessed through entry-points. A service runs on a service platform which allows organisations and individuals to serve and consume content through the Internet. There are various levels of services and various kinds of services offered.

Operational profile
Operational profile is a complete description of the work an SUT has to complete during a performance test. This term has the same meaning as workload term.

Performance characteristic
Performance characteristics are elaborated performance metrics derived from performance metrics. Examples: mean, standard deviation, maximum, minimum.

Performance measurement
A performance measurements is a measurement undertaken while running a performance test in order to derive the performance metrics.

Performance metric
A performance metric is a computation formula based on performance measurements of an SuT reported in a performance test report. This term is used interchangeably with metric term.

Performance requirement
A performance requirement is a requirement established at the design of a system with respect to its performance in terms of response time, capacity, maximal sustained number of users, etc.

Performance test
A performance test is a test with the purpose to verify the performance requirements expressed as DO

Performance test information model
The performance test information model defines the structure of the performance test.

Performance test parameter
A performance test parameter is a parameter whose value determines the behaviour of a performance test. Performance test parameters are used in order to configure the workload.

Performance test plan
A performance test plan is a technical documentation related to the execution of the performance test. It contains the description of the hardware which is used to run the test, the software versions, the test tools and the test schedule. Part of the test plan is also the performance test procedure which is specific to the selected type of performance test: volume, load, stress, benchmark, etc.

Performance test procedure
The performance test procedure comprehends the steps to be performed by a performance test.

Performance test report
A performance test report is a document that provides a full description of the execution of a performance test on a test system. This term is used interchangeably with the test report term.

Performance testing
Performance testing is the qualitative and quantitative evaluation of an SUT under realistic conditions to check whether performance requirements are satisfied or not.

Performance testing methodology
A performance testing methodology comprehends methods, patterns and tools to design, implement and execute performance tests.

Performance testing process
The performance testing process describes the steps from the definition of the performance requirements to the execution and the evaluation of the results of performance tests.

Robustness testing
Robustness testing is a kind of performance test which is executed over extended periods of time to validate the stability and reliability of an SUT. During short runs of load tests or

volume tests the system may behave correctly, but extending the testing period of time to a few hours, it may reveal problems of the inspected system.

Scalability testing
Scalability testing is a special kind of load testing, where the system is put under increasing load to determine how much more capacity a system has when the resources vary.

Scenario
A scenario defines a set of interactions to use an SUT service. Several scenarios may be associated to the same service. This term is used interchangeably with test scenario term.

Service
The service term refers to a telecommunication service offered by a telecoms provider to users. A user uses a service by interacting with the service platform, typically, through a sequence of messages.

Stateful session
A stateful test session keeps track of the state of communication between the entities involved in that session. A test system capable of stateful testing verifies that all messages are consistent with the state of the session.

Stateless
A stateless test session does not keep track of the state of communication between the entities involved in that session. A stateless test system does not verify the consistency of the messages in relation with the state of the session.

Stress testing
A stress test is a kind of performance tests which simulates the activities that are more stressful than the application is expected to encounter when delivered to real users.

System
System indicates the system to be tested. Since this thesis focuses on multi-service systems, the term system refers only to these types of systems. This terms is used interchangeably with the SUT term.

System load
A system load is a stream of protocol interactions presented to the SUT by the test system.

Test container
The test container is the host of test executables and manages the creation, configuration, communication and the removal of the parallel test components.

Test daemon
Test daemon are standalone processes installed on the test nodes (one test daemon per node) with the role to manage the test containers.

Test development
Test development is the concrete implementation of a test specification within a test framework. In this thesis, the TTCN-3 language is used for both, test specification and test development.

Test execution
Test execution is an automated process to apply test stimuli to an SUT while the behaviour of the SUT is monitored, and expected and actual behaviours are compared in order to yield a verdict.

Test harness
Test harness is the central part of a test execution platform which supplies the functionality to create, execute and validate tests. The elements of the test harness are: stubs, test drivers and test control systems.

Test node
A test node is a computing device involved in test execution. The concept covers anything one might use to run test activities such as general purpose PCs, specialised testing hardware with SUT specific connection hardware, or with protocol specific hardware stack or real-time computation systems. The test nodes are connected by physical communication channels. The connections may be of various types and technologies and typically include network connection devices such as routers or switches and UTP cables.

Test report
A test report is a document that provides a full description of the execution of a performance test on a test system. This term is used interchangeably with the performance test report term.

Test repository
A test repository is a storage mechanism providing versioning feature for the source files of a test suite.

Test scenario
A test scenario defines a set of interactions to use an SUT service. Several scenarios may be associated to the same service. This term is used interchangeably with scenario term.

Test session
A test session is the concrete interaction between a user and SUT and includes the concrete dialog information, e.g., session id, user ids.

Traffic model
A traffic model describes patterns for the request arrival rates with varying inter-arrival times and varying packet length.

Traffic set
Traffic set is a collection of test scenarios which are determined to be likely to co-occur in a real-world scenario. The scenarios need not come from the same use-case. Within a traffic set, each scenario has an associated relative occurrence frequency, interpreted as the probability with which it would occur in the course of the test procedure.

Traffic-time profile
Traffic-time profile is a function describing the average arrival rate as a function of elapsed time during a performance test.

Use-case
A use-case is a specification of a type of interaction between a TS and a SUT, corresponding to a mode of end-user behaviour.

User
A user is a person who consumes a service. The user uses a user equipment to access the service.

User equipment
A user equipment is a device used by a user to access and consume a service. This term is used interchangeably with the user term.

User pool
A user pool is a set of users which are intended to perform a similar task, e.g., run a particular type of test scenario.

User population
User population is the set of users involved in a performance test. Each user of a user population can be selected by random to be used in a test scenario.

Volume testing
A volume test is the kind of performance test which scales the number of users in order to find out which is the largest number of supported users by an SUT.

Workload
A workload is a complete description of the work an SUT has to complete during a performance test. In this thesis, a workload comprehends use-case, the traffic-time profile and the Traffic set.

Workload characterisation
Workload characterisation is the activity to select and define a workload with the goal to produce models that are capable of describing and reproducing the behaviour of a workload.

Acronyms

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
AS	Application Server
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitives
BCMP	BRAIN Candidate Mobility Management Protocol
CCF	Container Configuration File
CD	Coder/Decoder
CDMA	Code Division Multiple Access
CH	Component Handling
CP	Coordination Point
CSCF	Call Session Control Function
CTMF	Conformance Testing Methodology and Framework
D_MinStrContent	Data Representation with Minimal Structure Content
D_StrContent	Data Representation with Complete Structured Content
D_StringBuffer	Data Representation using String Buffer
D_StrPointers	Data Representation as Structure of Pointers
DNS	Domain Name System
DO	Design Objective
DOC	Design Objective Capacity
ETSI	European Telecommunications Standards Institute
GPS	Global Positioning System
GSM	Global System for Mobile Communications

GUI	Graphical User Interface
HSS	Home Subscriber Server
I-CSCF	Interrogating Call Session Control Function
IDL	Interface Description Language
IEC	International Electrotechnical Commission
IHS	Inadequately Handled Scenarios
IHSA	Inadequately Handled Scenario Attempt
IMS	IP Multimedia Subsystem
ISO	International Standards Organization
LG_MGenCtrl	Load Generation with Multiple Generators and Centralised Data
LG_MGenCtrl_Pull	Load Generation with Multiple Generators and Centralised Data using Pull Method
LG_MGenCtrl_Push	Load Generation with Multiple Generators and Centralised Data using Push Method
LG_MGenDectrl	Load Generation with Multiple Generators and Decentralised Data
LG_SGen	Load Generation with Single Generator
MTC	Main Test Component
OS	Operating System
P-CSCF	Proxy Call Session Control Function
P_Clusters	Population with Pool Clusters
P_MinClusters	Population with Minimal Number of Pool Clusters
P_SinglePool	Population with Single Pool
PA	Platform Adapter
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PTC	Parallel Test Component
QoS	Quality of Service (QoS)
R_MainThread	Receive with Main Thread
R_SepThreadPerSession	Receive with Separate Thread per Session
R_ThreadPool	Receive with Thread Pool
R_ThreadPool_Proactor	Receive with Thread Pool with Proactor
R_ThreadPool_Reactor	Receive with Thread Pool with Reactor

RFC	Request for Comments
RMI	Remote Method Invocation
RTP	Real-Time Transport Protocol
S-CSCF	Serving Call Session Control Function
S_MainThread	Send with the Main Thread
S_SepThreadPerRequest	Send with Separate Thread per Request
S_SepThreadPerSession	Send with Separate Thread per Session
S_ThreadPool	Send with Thread Pool
SA	System Adapter
SAPS	Scenario Attempts per Second
SE	Service Entity
SIMS	Simultaneous Scenarios
SIP	Session Initiation Protocol
SLA	Service Layer Agreements
SLF	Subscriber Location Function
SM	Session Manager
SM_GenHdl	State Machine with Generic Handler
SM_SpecHdl	State Machine with Specific Handler
SP	Service Provider
SUT	System Under Test
T_SepTT	Timer with Separate Timer Thread
T_Sleep	Timer with Sleep Operation
TA	Test Adapter
TC	Test Console
TCI	TTCN-3 Control Interfaces
Tcl/Tk	Tool Command Language/Toolkit
TE	Test Executable
TL	Test Logging
TM	Test Management
TRI	TTCN-3 Runtime Interfaces
TS	Test System
TTCN	Tree and Tabular Combined Notation
TTCN-2	Tree and Tabular Combined Notation, version 2
TTCN-3	Testing and Test Control Notation, version 3
UAC	User Agent Client
UAS	User Agent Server
UE	User Equipment
UH_InterleavedUPT	User Handler with Interleaved Users per Thread
UH_SeqUPT	User Handler with Sequential Users per Thread
UH_SingleUPT	User Handler with Single User per Thread

UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
UPSF	User Profile Server Function
UTP	Unshielded Twisted Pair
VoIP	Voice over IP
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Bibliography

[3GP06] 3rd Generation Parnership Project (3GPP). Technical Specification Group Services and System Aspects, IP Multimedia Subsystem (IMS), 23.228, Stage 2, V5.15.0, 2006.

[3GP08] 3rd Generation Parnership Project (3GPP). `http://www.3gpp.org`, 2008. Accessed 9 May 2008.

[AB.08] MySQL AB. MySQL 5.0. `http://www.mysql.com/`, 1995-2008. Accessed 20 Mar. 2008.

[Ack04] R. Ackerley. *Telecommunications Performance Engineering*. The Institution of Electrical Engineers, 2004. ISBN 0-86341-341-2.

[ACM08] ACM, Inc. The ACM Portal. `http://portal.acm.org/portal.cfm`, 2008. Accessed 9 May 2008.

[ÁCV02] G. Álvarez-Campana and B. Vázquez. A Simulation Tool for Dimensioning and Performance Evaluation of the UMTS Terrestrial Radio Access Network. In *Protocols and Systems for Interactive Distributed Multimedia: Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems, IDMS/PROMS 2002, Coimbra, Portugal, November 26-29, 2002. Proceedings*, 2002.

[Afu04] T.J.O. Afullo. Quality of service in telecommunications - the customer's perspective. In *AFRICON, 2004. 7th AFRICON Conference in Africa*, volume 1, pages 101–106, 2004. ISBN 0-7803-8605-1.

[AG08] SQS Software Quality Systems AG. SQS-TEST Professional version 8.0. `http://www.sqs.de/portfolio/tools/tools_sqstest_anwend.htm`, 2008. Accessed 15 Oct. 2007.

[AKLW02] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *Proceedings of the 3rd international workshop on Software and performance*, pages 17–24. ACM, 2002. Rome, Italy. ISBN 1-58113-563-7.

[AL93] A. Avritzer and B. Larson. Load testing software using deterministic state testing. *SIGSOFT Softw. Eng. Notes*, 18:82–88, 1993. ISSN 0163-5948.

[Alb03] S. T. Albin. *The Art of Software Architecture: Design Methods and Techniques*. John Wiley & Sons, Inc., 2003. ISBN 978-0-471-22886-8.

[Ale77] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN 0-19-501919-9.

[Alz04] Z. Alzamil. Application of the operational profile in software performance analysis. In *Proceedings of the 4th international workshop on Software and performance*, pages 64–68, Redwood Shores, California, 2004. ACM. ISBN 1-58113-673-0.

[Ama98] P. Amaranth. A Tcl-based multithreaded test harness. In *TCLTK'98: Proceedings of the 6th conference on Annual Tcl/Tk Workshop, 1998*, pages 8–8. USENIX Association, 1998. San Diego, California.

[Apa07] Apache Software Foundation. Apache JMeter. `http://jakarta.apache.org/jmeter/`, 1999-2007. Accessed 15 Oct. 2007.

[Apo04] D. Apostolidis. Handling of Test Components in Distributed Test Setups. Diploma Thesis. Technical University of Berlin, 2004.

[AW94] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 44–57. ACM, 1994. Seattle, Washington, United States. ISBN 0-89791-683-2.

[Bar02] S. Barber. FAQ - Performance & Load Testing. `http://www.sqaforums.com/showflat.php?Cat=0&Number=41861&an=0&page=0`, 2002. Accessed 30 Apr. 2008.

[Bar04a] S. Barber. Beyond performance testing part 1: Introduction. In IBM developerWorks. `http://www.ibm.com/developerworks/rational/library/4169.html`, 2004. Accessed 9 May 2008.

[Bar04b] S. Barber. Creating Effective Load Models for Performance Testing with Incomplete Empirical Data. In *Web Site Evolution, 2004. WSE 2004. Proceedings. Sixth IEEE International Workshop*, pages 51–59. IEEE Computer Society, 2004. ISBN 0-7695-2224-6.

[BBDD06] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Exploring the Effects of Hyper-Threading on Parallel Simulation. In *DS-RT '06: Proceedings of the 10th IEEE international symposium on Distributed Simulation and Real-Time Applications*, pages 257–260. IEEE Computer Society, 2006. ISBN 0-7695-2697-7.

[BCM04] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM, 2004. New York, NY, USA. ISBN 1-58113-873-3.

[BDZ03] M. Beyer, W. Dulz, and F. Zhen. Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains. In *12th Asian Test Symposium (ATS'03)*, volume 0, page 102. IEEE Computer Society, 2003. ISSN 1081-7735.

[BEA07] BEA Systems. Dev2Dev By developers, for developers. `http://dev2dev.bea.com/`, 2007. Accessed 9 May 2008.

[BG94] B. Baumgarten and A. Giessler. *OSI Conformance Testing Methodology and TTCN*. Elsevier Science Inc., New York, NY, USA, 1994. ISBN 0-444-89712-7.

[BHS+99] M. Born, A. Hoffmann, I. Schieferdecker, T. Vassiliou-Gioles, and M. Winkler. Performance testing of a TINA platform. In *Telecommunications Information Networking Architecture Conference Proceedings, 1999. TINA '99*, pages 273–278, 1999.

[BHS07] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley & Sons, Inc., 2007. ISBN 978-0-470-05902-9.

[Bin99] R. Binder. *Testing Object Oriented Systems. Models, Patterns and Tools.: Models, Patterns and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, chapter TTCN-3, page 659. Springer, August 2005. ISBN 978-3-540-26278-7.

[Bor07] Borland Software Corporation. SilkPerformer. `http://www.borland.com/us/products/silk/silkperformer/`, 2007. Accessed 15 Oct. 2007.

[Bur03] J. Buret. An overview of load test tools. `http://clif.objectweb.org/load_tools_overview.pdf`, 2003. Accessed 15 Oct. 2007.

[BZ 07] BZ Media LLC. Software Test & Performance. `http://www.stpmag.com/`, 2007. Accessed 9 May 2008.

[Cas04] J. P. Castro. *All IP in 3G CDMA Networks: The UTMS Infrastructure and Service Platforms for Future Mobile Systems*. John Wiley & Sons, Inc., 2nd edition, 2004. ISBN 0-470-85322-0.

[CGM05] G. Camarillo and M. A. Garcia-Martin. *The 3G IP Multimedia Subsystem (IMS). Merging the Internet and the Cellular Worlds*. John Wiley & Sons , Inc., 2rev edition, December 2005. ISBN 0-470-01818-6.

[Cit07] NEC and Penn State. CiteSeer.IST Scientific Literature Digital Library. `http://citeseer.ist.psu.edu/`, 2007. Accessed 9 May 2008.

[Cla07] Clarkware Consulting, Inc. JUnitPerf. `http://clarkware.com/software/JUnitPerf.html`, 2007. Accessed 15 Oct. 2007.

[Com07] Compuware Corporation. QALoad. `http://www.compuware.com/products/qacenter/qaload.htm`, 2007. Accessed 15 Oct. 2007.

[Cor07] AppPerfect Corporation. AppPerfect Load Tester version 9.5. `http://www.appperfect.com/`, 2002-2007. Accessed 20 Mar. 2008.

[CPFS07] S. Chakraborty, J. Peisa, T. Frankkila, and P. Synnergren. *IMS Multimedia Telephony Over Cellular Systems. VoIP Evolution in a Converged Telecommunication World*. John Wiley & Sons, Inc, 2007. ISBN 978-0-470-05855-8.

[Dai06] Z. R. Dai. *An Approach to Model-Driven Testing - Functional and Real-Time Testing with UML 2.0, U2TP and TTCN-3*. PhD thesis, Technical University of Berlin, 2006. Fraunhofer Verlag IRB ISBN: 3-8167-7237-4.

[Dav02] C. R. Davis. *IPSec. Tunneling im Internet*. MITP, 2002. ISBN 3-518-29155-6.

[DES08] G. Din, K.-P. Eckert, and I. Schieferdecker. A Workload Model for Benchmarking BPEL Engines. In *ICST '08: Proceedings of the International Conference on Software Testing Verification and Validation*. IEEE, 2008. Lillehammer, Norway.

[DF03] T. Dahlen and T. Fritzon. *Advanced J2EE Platform Development: Applying Integration Tier Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0-13-044912-1.

[Din06] G. Din. IMS Testing and Benchmarking Tutorial, The 2nd International FOKUS IMS Workshop. `www.fokus.fraunhofer.de/event/ims_ws_06/details.php?lang=en`, November 2006. Accessed 9 May 2008.

[DPB+06] J. Davidson, J. Peters, M. Bhatia, S. Kalidindi, and S. Mukherjee. *Voice over IP Fundamentals (2nd Edition) (Paperback)*. Cisco Press, February 2006. ISBN 1-5870-5257-1.

[DPE04] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. *SIGSOFT Softw. Eng. Notes*, 29:94–103, 2004.

[DPS07] G. Din, R. Petre, and I. Schieferdecker. A Workload Model for Benchmarking IMS Core Networks. In *Global Telecommunications Conference, 2007. GLOBECOM '07*, pages 2623–2627. IEEE, 2007. ISBN 978-1-4244-1043-9.

[DRCO05] J. D. Davis, S. E. Richardson, C. Charitsis, and K. Olukotun. A chip prototyping substrate: the flexible architecture for simulation and testing (FAST) . In *SIGARCH Comput. Archit. News*, volume 33, pages 34–43. ACM, 2005. New York, NY, USA. ISSN 0163-5964.

[DST04] S. Dibuz, T. Szabó, and Z. Torpis. BCMP Performance Test with TTCN-3 Mobile Node Emulator. In *TestCom*, volume 2978/2004 of *Lecture Notes in Computer Science*, pages 50–59. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21219-5.

[DTS06] G. Din, S. Tolea, and I. Schieferdecker. Distributed Load Tests with TTCN-3. In *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2006. ISBN 3-540-34184-6.

[Els07] Elsevier B.V. Elsevier. `http://www.elsevier.com`, 2007. Accessed 9 May 2008.

[EM02] S. Elnaffar and P. Martin. Characterizing computer systems' workloads. School of Computing, Queen's University, Canada, 2002.

[Emb07] Embarcadero Technologies, Inc. Embarcadero Extreme Test. `www.embarcadero.com/`, 2007. Accessed 15 Oct. 2007.

[Emp07] Empirix. e-Load. `www.scl.com/products/empirix/testing/e-load`, 2007. Accessed 15 Oct. 2007.

[Emp08] Empirix. Hammer. `http://www.empirix.com/products-services/v-load_test.asp`, 2008. Accessed 20 Mar. 2008.

[ER96] N. S. Eickelmann and D. J. Richardson. An evaluation of software test environment architectures. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 353–364. IEEE Computer Society, 1996. Berlin, Germany. ISBN 0-8186-7246-3.

[Eri04] Ericsson. IMS IP Multimedia Architecture. The value of using the IMS architecture, October 2004.

[ETS05] European Telecommunications Standards Institute (ETSI). Technical Specification TS 102 027-3: SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT), 2005. Sophia-Antipolis, France.

[ETS07a] European Telecommunications Standards Institute (ETSI). European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2007. Sophia-Antipolis, France.

[ETS07b] European Telecommunications Standards Institute (ETSI). European Standard (ES) 201 873-5 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI), 2007. Sophia-Antipolis, France.

[ETS07c] European Telecommunications Standards Institute (ETSI). European Standard (ES) 201 873-6 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI), 2007. Sophia-Antipolis, France.

[Fau07] D. Faught. Software Testing FAQ: Load and Performance Tools. `http://www.testingfaqs.org/t-load.html`, 2007. Accessed 9 May 2008.

[FG99] M. Fewster and D. Graham. *Software Test Automation*. Addison-Wesley Professional, 1st edition, August 1999. ISBN 0-201-33140-3.

[FMH93] W. Fischer and K. Meier-Hellstern. The Markov-modulated Poisson process (MMPP) cookbook. *Perform. Eval.*, 18(2):149–171, 1993.

[FOK06] Fraunhofer FOKUS. FOKUS Open Source IMS Core. `http://www.openimscore.org`, 2006. Accessed 15 Oct. 2007.

[FOK07] Fraunhofer FOKUS. IMS Benchmarking Project. `http://www.fokus.fraunhofer.de/IMSBenchmarking?lang=en`, 2006-2007. Accessed 15 Oct. 2007.

[Fok08] Fraunhofer fokus institute for open communication systems. Web site `http://www.fokus.fraunhofer.de`, 2008. Accessed 20 Mar. 2008.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1995. ISBN 0-201-63361-2.

[GHKR04] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using PEPA nets. In *SIGSOFT Softw. Eng. Notes*, volume 29, pages 13–23. ACM, 2004. 0163-5948.

[GHR+03] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42:375–403, June 2003.

[GJK99] M. Greiner, M. Jobmann, and C. Klüppelberg. Telecommunication traffic, queueing models, and subexponential distributions. *Queueing Syst. Theory Appl.*, 33(1-3):125–152, 1999.

[GMT04] Gerndt, Mohr, and Träff. Evaluating OpenMP Performance Analysis Tools with the APART Test Suite. In *Euro-Par 2004 Parallel Processing*, volume 3149/2004 of *Lecture Notes in Computer Science*, pages 155–162. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-22924-7.

[Gra00] J. Grabowski. TTCN-3 – A new Test Specification Language for Black-Box Testing of Distributed Systems. In *Proceedings of the 17th International Conference and Exposition on Testing Computer Software (TCS 2000), Theme: Testing Technology vs.Testers' Requirements*, 2000. Washington D.C.

[GTW03] J. Z. Gao, H.-S. J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software*. Artech House Publishers, August 2003. ISBN 1-58053-480-5.

[Har07] L. Harte. *IPTV Basics, Technology, Operation and Services (Paperback)*. Althos, 2007. ISBN 1-932813-56.

[HM01] S. Halter and S. Munroe. *Enterprise Java Performance*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, August 2001. ISBN 0-13-017296-0.

[HMR02] T. Halonen, J. Melero, and J. Romero. *GSM, GPRS and Edge Performance: Evolution Towards 3G/UMTS*. Halsted Press, New York, NY, USA, 2002. ISBN 0-470-84457-4.

[IBM07a] IBM developerWorks. developerWorks IBM's resource for developers. `http://www.ibm.com/developerworks/`, 2007. Accessed 9 May 2008.

[IBM07b] IBM. Rational Performance Tester v 7.0. `http://www.ibm.com/developerworks/rational/products/performancetester/`, 2007. Accessed 15 Oct. 2007.

[IEE07] Institute of Electrical and Electronics Engineers (IEEE). IEEE Xplore. `http://ieeexplore.ieee.org`, 2007. Accessed 9 May 2008.

[IET81] Internet Engineering Task Force (IETF) . RFC 791, IP: Internet Protocol. `http://www.ietf.org/rfc/rfc791.txt`, 1981.

[IET05] Internet Engineering Task Force (IETF). RFC 3261, SIP: Session Initiation Protocol. `tools.ietf.org/html/rfc3261`, 2005. Accessed 9 May 2008.

[Ins06] National Instruments. Designing next generation test systems - an in-depth user guide- developer zone - national instruments. `ftp://ftp.ni.com/pub/devzone/pdf/tut_3238.pdf`, 2006. Accessed 20 Mar. 2008.

[Int08] Intel Corporation. Web Site `http://www.intel.com/`, 2008. Accessed 20 Mar. 2008.

[Ipt07] Iptel.org. SIP Express Router. `http://www.iptel.org/ser/`, 2001-2007. Accessed 20 Mar. 2008.

[ISO94] ISO/IEC. Information technology - Open Systems Interconnection, Conformance testing methodology and framework. ISO/IEC 9646, 1994.

[Ixi07] Ixia. IxVoice. `http://ixiacom.com/products/display?skey=ixv_sip`, 2007. Accessed 15. Oct. 2007.

[Jai91] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley & Sons, Inc., April 1991. ISBN 0-471-50336-3.

[Jav08] Network World, Inc. Java World. `http://www.javaworld.com/`, 2006-2008. Accessed 9 May 2008.

[JE05] L. K. John and L. Eeckhout. *Performance Evaluation and Benchmarking*. CRC Press, September 2005. ISBN 0-8493-3622-8.

[Jeu99] G. C. Jeunhomme. *Measuring Telecommunication Infrastructure Needs and Demand*. PhD thesis, Massachusetts Institute of Technology, 1999. Accessed 9 May 2008.

[JG06] O. Jaques and R. Gayraud. SIPp. `http://sipp.sourceforge.net/`, 2006.

[JUn07] Object Mentor and Agile/XP. JUnit.org Resources for Test Driven Development. `http://www.junit.org/`, 2007. Accessed 15 Oct. 2007.

[KFN99] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 1999. ISBN 0-47135-846-0.

[KPvV00] A. C. Keskin, T. I. Patzschke, and E. von Voigt. Tcl/Tk: a strong basis for complex load testing systems. In *TCLTK'00: Proceedings of the 7th conference on USENIX Tcl/Tk Conference*, volume 7, pages 6–6. USENIX Association, 2000. Austin, Texas.

[KR07] R. Kreher and T. Ruedebusch. *UMTS Signaling: UMTS Interfaces, Protocols, Message Flows and Procedures Analyzed and Explained*. John Wiley & Sons , Inc., 2nd edition, February 2007. ISBN 978-0-470-06533-4.

[KSG+07] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394. IEEE Computer Society, 2007. ISBN 0-7695-3047-8.

[Lea99] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern*. Prentice Hall PTR, 2nd edition, November 1999. ISBN 0-201-31009-0.

[LF06] G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol*, 48:1172–1186, 2006.

[LGL+02] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 123–130. Australian Computer Society, Inc., 2002. Sydney, Australia. ISBN 0-909925-88-7.

[LKA04] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, Apr. 2004. ISBN 1-58488-397-9.

[LMPS04] S. Loveland, G. Miller, R. Prewitt, and M. Shannon. *Software Testing Techniques: Finding the Defects that Matter*. Charles River Media, 1 edition, October 2004. ISBN/SKU 1-58450-346-7.

[Ltd07] RadView Software Ltd. Webload professional version 8.0. `http://www.radview.com/product/description-overview.aspx`, 2007. Accessed 15 Oct. 2007.

[LyWW03] M. Lepper, B. Trancón y Widemann, and J. Wieland. TUB-TCI An Architecture for Dynamic Deployment of Test Components. In *Testing of Communicating Systems, 15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France, May 26-28, 2003, Proceedings*, Lecture Notes in Computer Science, pages 279–294. Springer, 2003. ISBN 3-540-40123-7.

[MA98] D. Menasce and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, May 1998. ISBN 0-13-693822-1.

[MAFM99] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 119–128. ACM, 1999. ISBN 1-58113-176-3.

[Men02a] D. A. Menascé. Load testing, benchmarking, and application performance management for the web. In *Int. CMG Conference*, pages 271–282, 2002.

[Men02b] D.A. Menasce. TPC-W: a benchmark for e-commerce. *Internet Computing. IEEE*, 6:83–87, 2002. ISSN 1089-7801.

[Mer07] Mercury. LoadRunner. `http://www.mercury.com/`, 2007. Accessed 15 Oct. 2007.

[Mic07] Microsoft TechNet. Web Application Stress. `http://www.microsoft.com/technet/archive/itsolutions/intranet/downloads/webtutor.mspx?mfr=true`, 2007. Accessed 15 Oct. 2007.

[MP02] D. McDysan and D. Paw. *ATM & MPLS Theory & Applications: Foundations of Multi-service Networking, 1st edition*. McGraw-Hill/Osborne, Berkley, California, USA, 2002. ISBN 0-07-222256-5.

[Neu04] H. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Universität Göttingen, 2004.

[NIS06] NIST/SEMATECH. e-Handbook of Statistical Methods. `http://www.itl.nist.gov/div898/handbook`, 2006. Accessed 15 Oct. 2007.

[Nix00] B.A. Nixon. Management of performance requirements for information systems. *IEEE Transactions on Software Engineering*, 26:1122–1146, 2000. IEEE. ISSN 0098-5589.

[NJHJ03] H. Q. Nguyen, B. Johnson, M. Hackett, and R. Johnson. *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems, Second Edition*. John Wiley & Sons, Inc., May 2003. ISBN 0-471-20100-6.

[oEI08] Institute of Electrical and Electronics Engineers (IEEE). IEEE 802.16 LAN/MAN Broadband Wireless LANS. `http://www.w3.org/XML/`, 2008. Accessed 20 Mar. 2008.

[Ope07] Opensourcetesting.org. Performance test tools. `http://www.opensourcetesting.org/performance.php`, 2007. Accessed 15 Oct. 2007.

[OR05] P. O'Doherty and M. Ranganathan. JAIN SIP Tutorial. `java.sun.com/products/jain/JAIN-SIP-Tutorial.pdf`, 2005. Accessed 9 May 2008.

[ORLS06] J. C. Okika, A. P. Ravn, Z. Liu, and L. Siddalingaiah. Developing a TTCN-3 test harness for legacy software. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 104–110. ACM Press, 2006. Shanghai, China. ISBN 1-59593-408-1.

[Pet06] R. Petre. Load Balancing and Parallelization Strategies within a TTCN-3 Test System. Diploma Thesis, Politechnica University of Bucharest, 2006.

[RBAS04] S. Rupp, F.-J. Banet, R.-L. Aladros, and G. Siegmund. Flexible Universal Networks - A New Approach to Telecommunication Services? *Wirel. Pers. Commun.*, 29(1-2):47–61, 2004. ISSN 0929-6212.

[SA07] S. S. Shirodkar and V. Apte. AutoPerf: an automated load generator and performance measurement tool for multi-tier software systems. In *Proceedings of the 16th international conference on World Wide Web*, pages 1291–1292. ACM, 2007. Banff, Alberta, Canada. ISBN 978-1-59593-654-7.

[SB04] J. Sommers and P. Barford. Self-configuring network traffic generation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 68–81. ACM, 2004. Taormina, Sicily, Italy. ISBN 1-58113-821-0.

[SDA05] I. Schieferdecker, G. Din, and D. Apostolidis. Distributed functional and load tests for web services. *Int. J. Softw. Tools Technol. Transf*, 7:351–360, 2005.

[SGG03] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts with Java*. John Wiley & Sons, Inc., 6 edition, October 2003. ISBN 0-471-48905-0.

[Shi03] J. Shirazi. *Java Performance Tuning*. O'Reilly Media, Inc., 2 edition, January 2003. ISBN 0-596-00377-3.

[SHK95] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. Wiley-IEEE Computer Society Press, 1995. ISBN 978-0-8186-6587-5.

[SJ01] S. Splaine and S. P. Jaskiel. *The Web Testing Handbook*. STQE Publishing, 2001. ISBN 0-970-4363-0-0.

[SK74] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Commun. ACM*, 17:127–133, 1974.

[SM08] Inc. Sun Microsystems. Java Platform, Standard Edition (Java 2 Platform). `http://java.sun.com/javase/`, 1994-2008. Accessed 20 Mar. 2008.

[Smi07] A. J. Smith. Workloads (creation and use). *Commun. ACM*, 50:45–50, 2007.

[SN90] W. R. Stevens and T. Narten. Unix network programming. *SIGCOMM Comput. Commun. Rev.*, 20(2):8–9, 1990.

[SNLD02] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SipStone: Benchmarking SIP Server Performance. `http://www.sipstone.org/files`, 2002. Columbia University, Ubiquity. Accessed 20 Mar. 2008.

[Spi06] Spirent Communications. Spirent Protocol Tester. `http://www.spirent.com`, 2006. Accessed 15 Oct. 2006.

[Spr07] Springer. SpringerLink. `http://www.springerlink.com/`, 2007. Accessed 9 May 2008.

[SPVG01] I. Schieferdecker, S. Pietsch, and T. Vassiliou-Gioles. Systematic Testing of Internet Protocols âĂŞ- First Experiences in Using TTCN-3 for SIP. In *5th IFIP Africom Conference on Communication Systems*, 2001. Cape Town, South Africa.

[SSR97] I. Schieferdecker, B. Stepien, and A. Rennoch. PerfTTCN, a TTCN language extension for performance testing. In *IFIP TC6 10th International Workshop on Testing of Communicating Systems (IWTCS'97) Proceedings*, pages 21–36. Chapman and Hall, 1997. Cheju Island, Korea. ISBN 0-412-81730-6.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., 2000. ISBN 0-471-60695-2.

[SSTD05] N. V. Stylianides, G. V. Stephanopoulos, G. S. Tselikis, and M. Dopher. Signaling Performance Trials and Evaluation Results on a GPRS Platform. *Information Systems Frontiers*, 7(2):129–140, 2005. Kluwer Academic Publishers. ISSN 1387-3326.

[Sta06] N. Stankovic. Patterns and tools for performance testing. In *Electro/onformation Technology, 2006 IEEE International Conference*, pages 152–157. IEEE, 2006. ISBN 0-7803-9593-X.

[SVG02] S. Schulz and T. Vassiliou-Gioles. Implementation of TTCN-3 Test Systems using the TRI. In *TestCom '02: Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, pages 425–442. Kluwer, B.V., 2002. ISBN 0-7923-7695-1.

[SVG03] I. Schieferdecker and T. Vassiliou-Gioles. Realizing Distributed TTCN-3 Test Systems with TCI. In Dieter Hogrefe and Anthony Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2003. ISBN 978-3-540-40123-0.

[SW01] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, 1st edition, September 2001. ISBN 0-201-72229-1.

[Sys07] Open Demand Systems. OpenLoad Tester V5.5 . `http://www.opendemand.com/openload/`, 2000-2007. Accessed 20 Mar. 2008.

[SZ02] N. Stankovic and K. Zhang. A distributed parallel programming framework. *IEEE Trans. Softw. Eng*, 28:478–493, 2002. IEEE Press. ISSN 0098-5589.

[Tan01] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001. ISBN 0-13-031358-0.

[Tec08] Testing Technologies. TTworkbench: an Eclipse based TTCN-3 IDE. `http://www.testingtech.com/products/ttworkbench.php`, 2008. Accessed 9 May 2008.

[TIS07] ETSI TISPAN. IMS/NGN Performance Benchmark, Technical Standard (TS) 186 008, 2007. Sophia-Antipolis, France.

[Tol05] S. Tolea. Dynamic Load Balancing For Distributed Test Systems. Diploma Thesis. Politechnica University of Bucharest, 2005.

[TSMW06] O. Teyeb, T. B. Sorensen, P. Mogensen, and J. Wigard. Subjective evaluation of packet service performance in UMTS and heterogeneous networks. In *Q2SWinet '06: Proceedings of the 2nd ACM international workshop on Quality of service & security for wireless and mobile networks*, pages 95–102. ACM, 2006. Terromolinos, Spain. ISBN 1-59593-486-3.

[TTe08] Testing Technologies IST GmbH. Web Site `http://www.testingtech.com`, 2008. Accessed 20 Mar. 2008.

[TTm08] TTmex - Distributed Test Platform. `http://www.testingtech.com/products/ttplugins_ttmex.php`, 2008. Testing Technologies. Accessed 20 Mar. 2008.

[VFS05] A. Vouffo-Feudjio and I. Schieferdecker. Test Patterns with TTCN-3. In *Formal Approaches to Software Testing*, volume 3395/2005 of *Lecture Notes in Computer Science*, pages 170–179. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-25109-5.

[VW98] F. I. Vokolos and E. J. Weyuker. Performance testing of software systems. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 80–87. ACM, 1998. Santa Fe, New Mexico, United States. ISBN 1-58113-060-0.

[VWM05] D. Vingarzan, P. Weik, and T. Magedanz. Design and Implementation of an Open IMS Core. In *Mobility Aware Technologies and Applications*, volume 3744, pages 284–293. Springer, 2005. ISBN 978-3-540-29410-8.

[W3C08] W3C. Extensible Markup Language. `http://www.w3.org/XML/`, 2008. Accessed 20 Mar. 2008.

[Web08] Web Performance, Inc. Web Performance Load Tester 3. `http://www.webperformanceinc.com/load_testing/`, 2008. Accessed 20 Mar. 2008.

[WJ03] B. Welch and K. Jones. *Practical Programming in Tcl & Tk, 4th Edition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, June 2003. ISBN 0-13-038560-3.

[WSG98] T. Walter, I. Schieferdecker, and J. Grabowski. Test architectures for distributed systems - state of the art and beyond. *Testing of Communicating Systems*, 11, 1998.

[WSP+02] C. Williams, H. Sluiman, D. Pitcher, M. Slavescu, J. Spratley, M. Brodhun, J. McLean, C. Rankin, and K. Rosengren. The STCL test tools architecture. *IBM System Journal*, 41(1):74–88, 2002. Accessed 9 May 2008.

[WV00] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26:1147–1156, 2000. ISSN 0098-5589.

[ZGLB06] L. Zhu, I. Gorton, Y. Liu, and N. B. Bui. Model driven benchmark generation for web services. In *Proceedings of the 2006 international workshop on Service-oriented software engineering*, pages 33–39. ACM, 2006. Shanghai, China. ISBN 1-59593-398-0.

Note: All Internet addresses have been verified at 9.05.2008.