Performance Test Design Process and Its Implementation Patterns for Multi-services Systems

George Din[1], Ina Schieferdecker[1,2], and Razvan Petre[1]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin
[2] Technical University Berlin, Franklinstr. 28/29, D-10623 Berlin

Abstract. Over the past years, the scope of telecommunication services has increased dramatically making network infrastructure-related services a very competitive market. Additionally, the traditional telecoms are combined with Internet technologies for providing a larger range of services. The obvious outcome is the increase of the number of subscribers and services demand. Due to this complexity, the performance testing of continuously evolving telecommunication services has become a real challenge and requires efficient and more powerful testing solutions. This ability highly depends on the performance test design and on the efficient use of hardware resources for test execution.

This paper proposes a performance testing methodology which copes with the characteristics mentioned above. The methodology consists of a set of methods and patterns, exemplified for the TTCN-3 language, to realize adequate performance tests for multi-service systems. The effectiveness of this methodology is demonstrated throughout a case study on IP Multimedia Subsystem (IMS) performance testing.

1 Introduction

The Service Providers (SP) are evolving their networks from legacy technologies to "fourth generation" technologies which involves: evolution of "traditional" wireline telecom standards to Voice over IP (VoIP) standards, evolution of GSM and CDMA networks to 3GPP/3GPP2 standards (e.g. UMTS), introduction of wireless LAN (WLAN) standards (e.g. IEEE 802.16), for both data and voice communications [17]. The current direction is to realize 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 Endpoints (UE), provided via a uniform interface.

The performance testing of telecommunication services raises a challenging problem: *how to create efficient tests to evaluate the performance of such systems?*. A typical multi-service system offers a number of services which can be accessed through entry-points [13]. The system consists of many sub-systems (hardware and software) communicating usually through more than one protocol. The service consumers are the end users which access the services through compatible devices called User-Endpoints (UEs). The service consumption is realized through communication protocols involving different types of transactions

K. Suzuki et al. (Eds.): TestCom/FATES 2008, LNCS 5047, pp. 135–152, 2008.

(e.g. authentication, charging, etc.). With the specification of IMS, the current telecommunication infrastructure is moving rapidly to a generic approach [5] to create, deploy and manage services, therefore we expect a large variety of services to be available soon in many tested systems.

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 is usually unpredictable since it depends on user preferences for services. As a result, the overall system load is a composition of instances from different services where each type of service contributes in a small proportion.

This paper elaborates a set of methods and patterns to design and implement efficient performance tests for systems which offer a large number of services and typically have to handle a large number of requests in short periods of time. We selected the TTCN-3 [10] technology to exemplify some of the patterns presented in the paper.

This paper is structured as follows. The next section presents several related works and it is followed by the description of the performance test design process for multi-service systems. Section 4 introduces the concrete elements. Section 5 discusses various implementation patterns. In Section 6 a case study which applies the performance test design methods to IP Multimedia System is presented. The paper ends with the conclusions section.

2 Related Work

There are various commercial and open source tools which can be used for performance testing of multi-service systems. Although there are many documents put out by software vendors, very few research papers present performance test tools design issues.

In [2] and [12] the Tcl/Tk [24] language is used to develop performance test frameworks. These papers explain the complex requirements of load testing and give a detailed overview for the extensive use of Tcl/Tk within the system. The design and implementation illustrate the code re-usability inherent in using Tcl as an application glue language.

The Faban tool [21] provides a framework to automate running of server benchmarks as well as a container to host benchmarks allowing new benchmarks to be deployed. Similarly, Weevil tool [23] is a generic automation tool for the deployment and execution of distributed workloads.

Another automated load generator and performance measurement tool for multi-tier software systems is Autoperf [18]. 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.

In [19] the Hammer [7], LoadRunner [15] and eLoad [6] are compared. The paper also motivates the implementation of an inhouse tool for performance testing.

The test tool is based on Visper middle-ware [20] which is written in Java and provides the primitives, components, and scalable generic services for direct implementation of architectural and domain specific decisions.

The work presented in this paper targets a missing gap in performance testing, namely the use of the TTCN-3 language for performance testing. Our platform enables TTCN-3 test engineers use the same language for both, functional and non-functional tests, but also benefit from the use of a dedicated testing language.

3 The Performance Test Design Process

Though performance testing is a common topic among people and organizations, the research does not address performance testing of multi-service systems at a general level but rather targets only for specific types of applications e.g. web applications [14]. The methodology presented in this paper is based on the performance test design process which is depicted in Figure 1. This process refines the general process described in [9] as applied to multi-service systems.

Performance Requirements. The process starts with the selection of *performance requirements*. In this step the test engineer has to identify which features characterize 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 utilization. A detailed view on performance requirements selection is provided in [3] where a performance requirements framework integrates and catalogues performance knowledge and the development process.

Workload. The next step is to define the *workload*. The workload comprehends the performance testing focus points, test interfaces and scenarios of interest. The

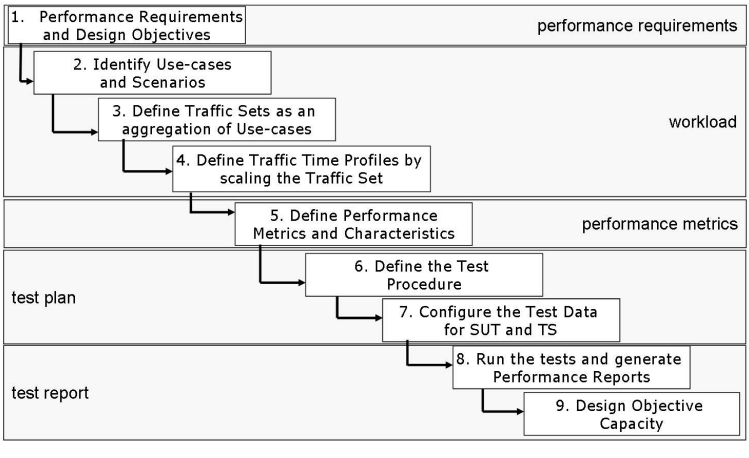

Fig. 1. The Performance Test Design Process

test engineer has to identify for each scenario the sequence of actions the test system should interchange with the (System under Test) SUT, interaction protocols and input data sets. 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.

Performance Metrics. In a third step, the performance metrics have to be defined. Two types of metrics are regarded: global metrics (similar to other related works [11] e.g. CPU, MEM, fail rate, throughput etc) and scenario related metrics (defined for each scenario e.g. error rate, latency).

Performance Test Plan. The workloads are documented in a *performance test plan* which contains the 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: volume, load, stress, benchmark etc.

Performance Test Report. A test report is a document, with accompanying data files, that provides a full description of an execution of a performance test. The test results should present the computed metrics and data sets in the form of charts, graph or other visual format.

4 Workload Elements

4.1 Use-Cases and Test Scenarios

The first step in the workload creation is to identify the *use-cases* and, for each use-case, select multiple *test scenarios*. A use-case is associated to one service and define interaction models between one or more users and the SUT (e.g. voice call, conference call).

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 we have to answer 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. A representative workload is realized when the selection of scenarios cover all possible situations: successful, fail, abandoned and rejected scenarios.

An example of a test scenario is depicted in Figure 2. This example presents the interaction between two User Endpoints (UE) and the SUT (all entities of the SUT are represented as a single box). The interaction is based on Request/Response transactions. Each response has to be received within a limit of time. This time is modeled as a timer which measures the time spend between the request and response. If the response is not received within the expected time, the transaction runs into a fail situation.

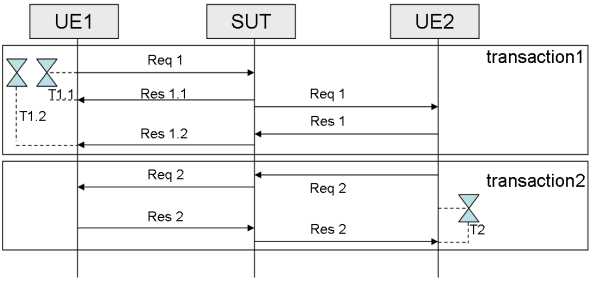

Fig. 2. Scenario Flow Example

4.2 Design Objectives

The performance requirements concern error-rates and delays. They are evaluated separately for each test scenario on top of the collected measurements. For each performance requirement a *design objective* (DO) is defined as a threshold value which is then used to compare the metrics. Examples of DOs are:

- *latency design objectives*: define the maximal time required to get a message through the SUT network from a caller UE1 to a callee UE2.
- *transaction round-trip time design objectives*: define the maximal time required to complete a transaction (including all messages).
- *error rates*: defines the threshold for allowed percentage of errors out of the number of instantiated scenarios.

4.3 Traffic Set Composition

The test system applies a workload to the SUT which consists of the traffic generated by a large number of individual simulated UEs. A conceptual question is how to allow testers define compositions of scenarios. The concept to cover this aspect is called *traffic set*.

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 test. This frequency indicates how often a scenario should be instantiated during the complete execution of the performance test.

To avoid constant load intensities (in reality the load is random), 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 characterizes the evolution of the average arrival rate as a function of time over the duration of the test procedure. An example of such an arrival process is the Poisson process [16] employed often in simulations of telecommunication traffic.

4.4 Traffic-Time Profile

A further concept is the *traffic-time profile* used to describe the workload intensity. The traffic-time profile defines the average *scenario attempt arrival rate* as

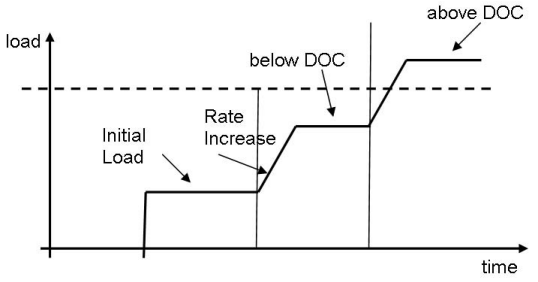

Fig. 3. Stair-Step Traffic-Time Profile

a function of elapsed time during a performance test. It should be defined in such a manner that, for a given scenario attempt arrival rate, sufficient samples are generated that metrics can be collected with an appropriate confidence bound.

A common traffic-time profile is the stair-step profile which is the one based on the *stairstep* shape (see Figure 3). The width of the stairstep is such chosen to collect sufficient samples at a constant average scenario arrival rate.

4.5 Scenario Based Performance Metrics

Based on the design objectives, performance metrics can be defined at scenario level.

Pass/Fail Metrics. The scenario attempts can be sorted into *passed* and *failed* scenarios. The number of failed versus passed scenarios out of the total attempts, is a metric which can be defined for each scenario. This metric is usually reported as a time based shape of fails or passes per second.

Transmission Latency. This metric is used to compute the average latency of the SUT at processing a certain event (*transmission latency*).

Round Trip Time. This metric is used to compute the average time needed to execute a transaction (*round-trip time*).

4.6 Global Performance Metrics

Besides the metrics computed per scenario, we also introduce global metrics to characterize the overall test execution. We classify them into: *resource usage metrics* and *troughput metrics*.

Resource usage metrics. These metrics capture the resource consumption of the SUT over time and may help identify *when* (i.e. at which load, after how much time) the SUT runs out of resources or starts failing.

Throughput metrics. The throughput metrics are the message rates and error rates. They are computed globally (or per use-case) for all scenarios. We propose a minimal set of metrics which any performance test should report.

- **SAPS.** Scenario Attempts Per Second (SAPS) metric represents the average rate at which scenarios are attempted.
- **SIMS.** SIMultaneous Scenarios (SIMS) counts the number of scenarios 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.
- **%IHS.** The *Inadequately Handled Scenarios*(IHS) metric is the ratio of inadequately handled scenarios out of the total number of attempted scenarios. A scenario is considered inadequately handled when it does not respect the specified call flow.

4.7 Design Objective Capacity Definition

To be able to compare between different versions, hardware, products, etc., a *comparison criterion* is needed. In our methodology we define a global performance indicator called *Design Objective Capacity* (DOC) as the number which characterizes the overall performance of an SUT. This number is defined as the load rate for which the SUT still can handle the load for certain quality conditions. Increasing this load intensity would automatically affect the SUT to exceed the design objectives.

5 Performance Test Implementation Patterns

The test behavior is realized as a collection of parallel threads. Most operating systems provide features enabling a process to contain multiple threads of control [22]. The performance test systems use lots of threads at execution. There are threads specific to the testing purpose 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. A type of a thread may be instantiated for an arbitrary number of times. For each type of action, more than one threads can be instantiated in order to increase the parallelism.

The implementation patterns described in the following refer to the way how different types of threads are interacting with each other, how they share the data and how to manipulate the number of threads of different scopes for gaining the best performance.

5.1 Data Repository Representation

The interaction between a test behaviour thread and the user's data repository is depicted in Figure 4. 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. 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.

A simple API to manage the user repository is presented in Listing 1. It consists of very simple operations to extract or update the information. Typical

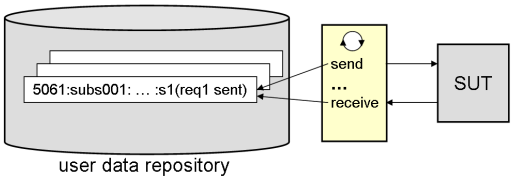

Fig. 4. User State Handling within a Thread

Listing 1. Repository Access API through external functions

```
external getAvailable() return userid;                                    1
external setState(userid u, state s);
external validateState(userid u, newstate ns) return boolean;
external makeAvailable(userid u);
```

operations are `getAvailable()` to get an available user to create a 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.

5.2 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. From the implementation point of view, the user state machine can be realized either in a specific way or in a generic way. We discuss these two approaches.

Specific Handler (SM_SpecHdl). In this pattern 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 avoids synchronization times with the rest of threads. As far as the implementation of this design is concerned, 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 be also simulated locally within the thread. 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.

Unfortunately, for large number of users, this pattern is not practical since many threads have to be created [4]. The most used operating systems (based on Unix and Windows) encounter all serious problems under conditions involving a tremendous number of threads.

In Listing 2, the specific handling mechanism is exemplified in the TTCN-3 language. The function gets a `userid` as start parameter to *personalize* it to a

Listing 2. Specific Event Handling

```
function user(id userid) runs on UserComponent {                    1
 var stateType state := state1;
 var requestType req; var responseType resp;
 alt {
  [state == state1] p.receive(requestType1(userid))
        -> value req {                                               6
    if(match(req.field1, expectedValue) {
      state := state2;
      resp := responseTypeTemplate;
      resp.field2:=req.field2; resp.field3:=req.field3+1;
      p.send(resp);                                                 11
      repeat;
    }
    else {state = failureState; makeAvailable(userid); stop;}
  }
  [state == state2] p.receive(requestType2(userid)) {               16
    state := finalState; makeAvailable(userid); stop; }
  [] p.receive {
    state = failureState; makeAvailable(userid); stop;}
 } //end alt
} // end function                                                   21
```

particular user. As long as the function implements a specific user behaviour, the *state* variable (representing the state of the user) can be defined in the function as a local variable. The events are received by the port **p** and are handled by an **alt** block. Each event type is caught by a receive statement. The template must contain matching constraints for the `userid` given as parameter to the function. This is simple to achieve by parameterizing the template with the `userid`.

In the example, we define three possible branches. 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 the branch. 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. For a non-valid message, the test system has to set the state of the user to `failureState` and stop the component.

Generic Handler (SM_GenHdl). The *specific event handler* approach suffers of performance problems when too many threads are created. Therefore, a better solution is to create less threads by using one thread for more than one user. This way, the platform scales obviously better than in the previous approach.

This pattern requires a global data management and the state machine is rather a message processor. The functionality of this pattern is depicted in Figure 5. When new messages are received, the message processor identifies the user to which the new message belongs to and updates its status in the required way.

This pattern has the advantage that a handler can be used to process an arbitrary number of users in parallel. As long as the user data is stored outside the thread, the thread does not keep track of 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 can be

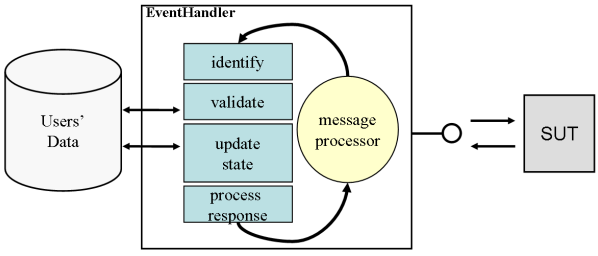

Fig. 5. Generic Event Handler

Listing 3. Generic Event Handling

```
function userHandler() runs on UserComponent {
 var requestType req; var responseType resp;
 alt {
  [] p.receive(requestType1) -> value req                                    4
  {
    if(not validState(req.userid, req)) {
      setState(req.userid, failureState);
      makeAvailable(req.userid); repeat;
    }                                                                        9
    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;                14
    }
    else { setState(req.userid, failureState);
      makeAvailable(userId); repeat;
    }
  }                                                                          19
  [] p.receive(requestType2) -> value req {
    if(not validState(req.userid, req)) {
      setState(req.userid, failureState);
      makeAvailable(req.userid); repeat;
    }                                                                        24
    setState(req.userid, finalState);
    makeAvailable(req.userid); repeat;
  }
  [] p.receive { setState(req.userid, failureState);
    makeAvailable(userid); repeat;                                           29
  }
 } // end alt
} // end function
```

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 for any user.

Listing 3 provides an example of a generic handler in TTCN-3. It basically rewrites the example from Listing 2 in a generic way. In the first example, the component has one port which is used for the communication with the SUT. In the second example, the same port is used for all users simulated by the component by using the **to address** statement. This concept optimizes the number of ports which require a lot of resources.

Next, instead of using the guard conditions, we introduce the **validState()** function to check the state of a user in a repository. The validation is realized

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 test system 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 parameterized 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.

5.3 Patterns for Thread Usage in User Handling

In the above section we discussed the two patterns to realize a user state machine. Both approaches have advantages and disadvantages. We present three more patterns regarding the thread design and usage. These patterns base 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.

Single User Behaviour per Thread (UH_SingleUPT). The easiest way to implement a user is to create an instance of a thread simulating only that user, i.e. the *single user per thread* pattern. This pattern is suitable to the *state machine specific* handling approach (SM_SpecHdl) but can also be combined with the generic approach. However, the creation, the start and the termination of threads are very expensive operations with respect to CPU. Therefore, this pattern will is not suitable for an efficient design.

Sequential User Behaviours per Thread (UH_SeqUPT). This pattern considers the *reusability* of a thread for new users. This method requires that the user identities are stored outside the thread and are loaded into the thread only when the user has to become active. 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.

Interleaved User Behaviours per Thread (UH_InterleavedUPT). An even better approach to use threads is to interleave user behaviours at the same time on the same thread. This pattern may be seen as an extension of the previous thread with the addition that users are handled at the same time. In this way, the thread is able to simulate in parallel an arbitrary number of users.

5.4 Pattern for Timer Implementation

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. Additionally, the user interaction

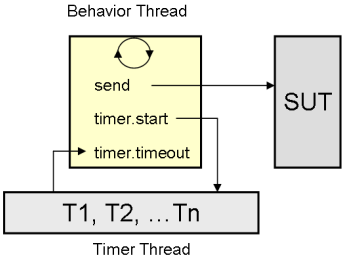

Fig. 6. Use of a Timer Thread for Timer realization

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 our test methodology. The test system has to validate if these time constraints are fulfilled.

In our approach, a timer thread manages all timers involved in test scenarios. 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 happen simply 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 out of the scheduling queue.

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 used in parallel for different kinds of timers.

5.5 Patterns for Sending Messages

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, sometimes it is not convenient to let a *state machine handling thread* spend too much time for these operations. We propose a concept in which a separate thread is instantiated for the 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. We distinguish four different ways to realise this concept.

Thread per Request (S_SepThreadPerRequest). The *thread per request pattern* requires that each send request is handled by a separate thread of control. This pattern is useful for test systems that simulate multiple users that handle long-duration request/response (such as database queries). It is less useful for short-duration requests due to the overhead of creating a new thread for each request.

Thread per Session (S_SepThreadPerSession). This pattern is a variation of the *thread per request pattern* that compensates the cost of spawning the thread across multiple requests. This pattern handles each user simulated by the test system in a separate thread for the duration of the session. It is useful for test systems that simulate multiple users that carry on long-duration conversations.

Send Thread Pool Pattern (S_ThreadPool). A number of threads are created for all message sending tasks, usually organized in a queue. These threads are shared between multiple test behaviour threads. 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 new tasks are available.

5.6 Patterns for Messages Receiving

Similar to sending of messages, there are several patterns to implement the receiving operations. For receiving of messages, the receive thread creates a message queue and reads from it whenever something is enqueued.

Thread per Session (R_SepThreadPerSession). In this pattern, a main thread deals with the main flow of actions (i.e. user behaviour) and another thread handles the events from SUT. The second thread only waits for SUT responses and notifies the main thread in case something is received.

Thread Pool for SUT Responses (R_ThreadPool). To simplify the thread management, test platforms organize their threads with thread pools. Since any

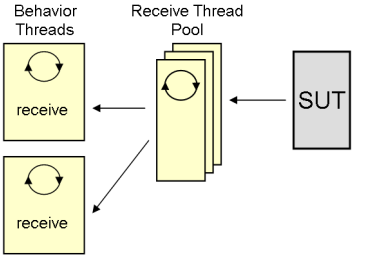

Fig. 7. Receive Thread Pool

main behaviour thread requires a second thread for checking SUT replies, we can share one such thread 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 approach requires also an identification mechanism, usually called *event demultiplexor*, between the received messages and the main threads (see Figure 7). The event demultiplexor is an object that dispatches 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.

5.7 Data Encapsulation Patterns

The content of a message is usually not handled in its raw form but through a message 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. We discuss three strategies to encapsulate and access the content of a message.

String-Buffer Pattern (D_StringBuffer). 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). 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. Unfortunately, the string processing costs a lot of CPU time. Therefore, the pattern should not be used for big messages.

Structured Representation of Message Content (D_StrContent). This pattern requires that the content of the message is represented as a tree structure (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. These operations may cost a lot of CPU time but the tree representation offers a lot of flexibility to easily access every piece of information.

Structured Representation of Pointers to Message Content (D_StrPointers). In this pattern, a tree structure contains pointers to the locations where the content can be accessed in the raw-message. This pattern also requires a codec but it can be implemented efficiently since the only task is to identify data at given locations in the raw-message. This pattern can be used even more efficiently when the locations are predefined e.g. by using fixed sizes for the fields' lenghts.

6 Case Study

The test patterns have been evaluated throughout a case study on IMS performance testing. IMS is a standardised architecture [1] to support a rich set of

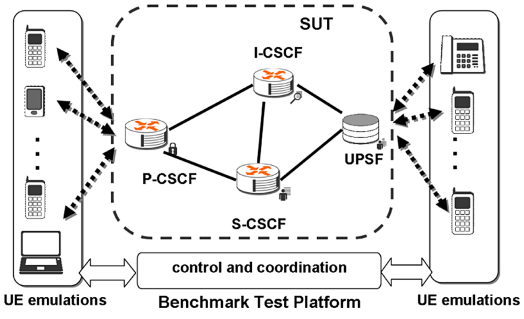

Fig. 8. Test System vs. System under Test

services and make them available to end users. The multitude of offered services and the complex call flows of the interactions between users and the IMS network, make IMS an excellent candidate to apply the methodology.

The SUT. The performance tests focus on the Session Control Subsytem (SCS) (see Figure 8). It consists of the three main (Call Session Control Function) CSCF elements: Proxy, Serving and Interrogating, and the (User Profile Server Functions) UPSF. The traffic set consists of scenarios which belong to the most common IMS use cases that are encountered the most in the real life deployments: *Registration and de-registration*, covering five scenarios, *Session set-up or tear-down*, covering twelve scenarios, *Page-mode messaging*, covering two scenarios.

The Test System Implementation. For the implementation of the test system we selected the TTCN-3 language. In TTCN-3, a thread corresponds to a *test component* and the thread functionality is implemented as a *function*. The load is generated by a specialised load generator component. Each call created by the load generator is associated to an `EventHandler` component, which handles all required transactions for that call. 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.

Along the case study development we improved step by step the performance of the test system by combining the patterns described in the paper. We illustrate these results in the following. Since the test system and the IMS SUT implementation were developed almost at the same time, the test system developers had to come with new solutions to increase the performance in order to compete the continuously improving performance of the SUT.

At the beginning, the test system was based on the *Specific Handler* design pattern (SM_SpecHdl) and on the Single User Behavior per Thread (UH_SingleUPT). The receiving of the messages was based on the *Thread per Session* (R_SepThreadPerSession) pattern while the data encapsulation was based on the structured representation of the message content (D_StrContent). This was the first draft of the implementation, the overall performance of

Table 1. Results 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

the test system was not amazing, but it was a very easy adaptation from a functional test suite.

The first improvement was related to the mechanism used for receiving messages. We changed to a thread pool approach for handling the received messages (R_ThreadPool) and the performance of the test system 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 we needed to test the maximal capacity of the SUT. By using this approach the test system 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 we create fewer threads then before.

In the early stages only 5 scenarios were supported, but in the end the traffic set consists of 20 different scenarios. This amount on different scenarios made that the Specific Handler pattern was not suitable anymore, thus we decided to adopt the Generic Handler pattern. The Generic Handler proved to be more flexible and easier to maintain and also more suitable to satisfy 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 we expect a remarkable increase of the performance to more than 300% out of the current performance.

Experimental Work. The performance test has been applied to different hardware configurations in order to illustrate how the the test can be used for performance comparison. The implementation for the IMS architecture used as SUT within the case study is OpenIMSCore [8], the open source implementation of Fraunhofer FOKUS.

Table 1 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.

The experiments revealed that many parameters have a great impact on the SUT performance. Among these parameters are: traffic set composition, number of users, duration of the test. Obiviously, changing the traffic composition will automatically change the demand of resources on SUT side. The duration is actually influenced by the number of total calls created along the test. For different loads but same duration, the test system creates different numbers of calls. With respect to the number of users, another experiment comparing the SUT performance with 5000 users respectively 10000 users, shown that the SUT performance is 40% worse when running the test with 10000 users.

7 Conclusions

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. This paper presented a methodology for performance testing of telecommunication systems. The topic embraces the challenges of the nowadays telecommunication technologies. The main outcome is the method to create workloads for performance testing of such systems. It takes into account many factors: use cases, call flows, test procedures, metrics and performance evaluation. The design method ensures that the workloads are realistic and simulate the conditions expected to occur in real-live usage.

Within the case study, we designed and developed a performance test suite capable of evaluating the performance of IMS networks for realistic workloads. The TTCN-3 language has been selected to allow a more concrete illustration of the presented concepts. Along implementation of the test framework we pursuit the efficiency related aspect which determined us to experiment with various patterns to design and implement performance test systems.

References

1. 3GPP. Technical Specification Group Services and System Aspects, IP Multimedia Subsystem (IMS), Stage 2, V5.15.0, TS 23.228 (2006)
2. Amaranth, P.: A Tcl-based multithreaded test harness. In: Proceedings of the 6th conference on Annual Tcl/Tk Workshop, San Diego, California, vol. 6, p. 8. USENIX Association (1998)
3. Nixon, B.A., Nixon, B.A.: Management of performance requirements for information systems. Transactions on Software Engineering 26, 1122–1146 (2000)
4. Born, M., Hoffmann, A., Schieferdecker, I., Vassiliou-Gioles, T., Winkler, M.: Performance testing of a TINA platform. In: Telecommunications Information Networking Architecture Conference Proceedings, TINA 1999, pp. 273–278 (1999)
5. Camarillo, G., Garcia-Martin, M.A.: The 3G IP Multimedia Subsystem (IMS). Merging the Internet and the Cellular Worlds, 2nd edn. Wiley & Sons, Chichester (2005)
6. Empirix. e-Load (2007), `http://www.scl.com/products/empirix/testing/e-load`
7. Empirix. Hammer (2007), `http://www.empirix.com/`

8. Fraunhofer FOKUS. FOKUS Open Source IMS Core (2006)
9. Gao, J.Z., Tsao, H.-S.J., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House Publishers (August 2003) ISBN 1580534805
10. Grabowski, J., Hogrefe, D., Rethy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks 42, 375–403 (2003)
11. Jain, R.K.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, Chichester (1991)
12. Keskin, A.C., Patzschke, T.I., von Voigt, E.: Tcl/Tk: a strong basis for complex load testing systems. In: Proceedings of the 7th conference on USENIX Tcl/Tk Conference, Austin, Texas, vol. 7, p. 6. USENIX Association (2000)
13. McDysan, D., Paw, D.: ATM & MPLS Theory & Applications: Foundations of Multi-service Networking. Mcgraw-Hill Professional, New York (2002)
14. Menascé, D.A., Almeida, V.A.F., Fonseca, R., Mendes, M.A.: A methodology for workload characterization of e-commerce sites. In: EC 1999. Proceedings of the 1st ACM conference on Electronic commerce, pp. 119–128. ACM, New York (1999)
15. Mercury. Mercury LoadRunner (2007), `http://www.mercury.com/`
16. NIST/SEMATECH. e-Handbook of Statistical Methods (2006)
17. Rupp, S., Banet, F.-J., Aladros, R.-L., Siegmund, G.: Flexible Universal Networks - A New Approach to Telecommunication Services? Wirel. Pers. Commun. 29(1-2), 47–61 (2004)
18. Shirodkar, S.S., Apte, V.: 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, Banff, Alberta, Canada, pp. 1291–1292. ACM Press, New York (2007)
19. Stankovic, N.: Patterns and tools for performance testing. In: Electro/information Technology, 2006 IEEE International Conference, pp. 152–157 (2006)
20. Stankovic, N., Zhang, K.: A distributed parallel programming framework. IEEE Trans. Softw. Eng. 28, 478–493 (2002)
21. SUN. Faban (2007), `http://faban.sunsource.net/`
22. Tanenbaum, A.: Modern Operating Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
23. Wang, Y., Rutherford, M.J., Carzaniga, A., Wolf, A.L.: Automating experimentation on distributed testbeds. In: ASE 2005. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 164–173. ACM, New York (2005)
24. Welch, B., Jones, K.: Practical Programming in Tcl and Tk, 4th edn. Prentice Hall PTR, Englewood Cliffs (2003)