Distributed Traffic Load Scheduler based on TITANSim for System Test of a Home Subscriber Server (HSS)

NIRANJANAN KALAICHELVAN

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Distributed Traffic Load Scheduler based on TITANSim for System Test of a Home Subscriber Server (HSS)

Niranjanan Kalaichelvan

nirkal@kth.se

Examiner: Prof. Gerald Q. "Chip" Maguire Jr., KTH
Industry supervisor: Peter Dimitrov, Ericsson AB

This thesis project was carried out at and funded by Ericsson AB, Stockholm, Sweden

Abstract

The system test is very significant in the development life cycle of a telecommunication network node. Tools such as TITANSim are used to develop the test framework upon which a load test application is created. These tools need to be highly efficient and optimized to reduce the cost of the system test. This thesis project created a load test application based on the distributed scheduling architecture of TITANSim, whereby multiple users can be simulated using a single test component. This new distributed scheduling system greatly reduces the number of operating system processes involved, thus reducing the memory consumption of the load test application; hence higher loads can be easily simulated with limited hardware resources.

The load test application used for system test of the HSS is based on the central scheduling architecture of TITANSim. The central scheduling architecture is a *function test* concept, where every user is simulated by a single test component. In the *system test* several thousand users are simulated by the test system. Therefore, the load application based on central scheduling architecture uses thousands of test components leading to high memory consumption in the test system. In this architecture, the scheduling of test components is centralized which results in a lot of communication overhead within the test system, as thousands of test components communicate with a *master scheduling* component during the test execution.

On the other hand, in the distributed scheduling architecture the scheduling task is performed *locally* by each test component. There is no communication overhead within the test system. Therefore, the test system is highly efficient. In the distributed scheduling architecture the traffic flow of the simulated users are described using the Finite State Machines (FSMs). The FSMs are specified in the configuration files that are used by the test system at run time. Therefore, implementing traffic cases using the distributed scheduling architecture becomes simpler and faster as there is no (TTCN-3) coding/compilation.

The HSS is the only node (within Ericsson) whose system test is performed using the central scheduling architecture of TITANSim. The other users (nodes) of TITANSim are using the distributed scheduling architecture for its apparent benefits. Under this circumstance, this thesis project assumes significance for the HSS. When a decision to adapt the distributed scheduling architecture is made for the system test of the HSS, the load application created in this thesis project can be used as a model, or extended for the migration of the test modules for the HSS from the central scheduling architecture to the distributed scheduling architecture.

By creating this load application we have gained significant knowledge of the TITANSim framework; most importantly, the necessary modifications to the TITANSim framework required to create a distributed scheduling architecture based load application for the HSS. The load application created for this project was used to (system) test the HSS by generating load using real system test hardware. The results were analytically compared with the test

results from the existing load application (which is based on the central scheduling architecture). The analysis showed that the load application based on distributed scheduling architecture is efficient, utilizes less test system resources, and capable of scaling up the load generation capacity.

Key words: FSM, scheduling, system test, TITANSim, test components

Sammanfattning

Systemet test är mycket betydelsefullt i utvecklingen livscykeln för ett telenät nod.Verktyg som TITANSim används för att utveckla testet ram på vilken ett belastningsprov program skapas. Dessa verktyg måste vara mycket effektiv och optimerad för att minska kostnaderna för systemet testet. Detta examensarbete skapat ett program belastningsprov bygger på distribuerad schemaläggning arkitektur TITANSim, där flera användare kan simuleras med hjälp av ett enda test komponent. Det nya distribuerade schemaläggning systemet minskar kraftigt antalet operativsystem inblandade system processer, vilket minskar minnesförbrukning av lasten testprogram, därav högre belastningar kan enkelt simuleras med begränsade hårdvara resurser.

Lasten testa program som används för systemtest av HSS är baserad på den centrala schemaläggning arkitektur TITANSim. Den centrala schemaläggning arkitektur är ett funktionstest koncept, där varje användare simuleras med ett enda test komponent. I systemet testa flera tusen användare är simulerade av testsystemet.Därför använder belastningen program baserat på centrala schemaläggning arkitektur tusentals testa komponenter leder till hög minnesförbrukning i testsystemet.I denna arkitektur är schemaläggning av test komponenter centraliserad vilket resulterar i en mycket kommunikation overhead inom testsystem, som tusentals testa komponenter kommunicerar med en mästare schemaläggning komponent under testexekvering.

Å andra sidan, i den distribuerade schemaläggning arkitekturen schemaläggning uppgiften utförs lokalt av varje test komponent. Det finns ingen kommunikation overhead i testsystemet. Därför är testsystemet mycket effektiv. I distribuerad schemaläggning arkitekturen trafikflödet av simulerade användare beskrivs med Finite State Machines (FSMs). Den FSMs anges i konfigurationsfiler som används av testsystemet vid körning. Därför genomföra trafiken fall med distribuerad schemaläggning arkitektur blir enklare och snabbare eftersom det inte finns någon (TTCN-3) kodning / sammanställning.

HSS är den enda nod (inom Ericsson) vars system test utförs med hjälp av den centrala schemaläggningen arkitektur TITANSim. Den andra användare (noder) i TITANSim använder distribuerad schemaläggning arkitektur för sina uppenbara fördelar. Under denna omständighet, förutsätter detta examensarbete betydelse för HSS. När ett beslut att anpassa distribuerad schemaläggning arkitektur är gjord för systemet test av HSS, kan belastningen program som skapats i detta examensarbete kan användas som en modell, eller förlängas för migration av testet moduler för HSS från den centrala schemaläggningen arkitektur för distribuerade schemaläggning arkitektur.

Genom att skapa denna belastning ansökan har vi fått stor kunskap om TITANSim ramen, viktigast av allt, de nödvändiga ändringar av TITANSim ramverk som krävs för att skapa en

distribuerad schemaläggning arkitektur baserad belastning ansökan för HSS. Lasten program som skapats för detta projekt har använts för att (system) testa HSS genom att generera last använda riktiga maskinvarusystem test. Resultaten analytiskt jämfört med provresultaten från den befintliga belastningen ansökan (som är baserad på den centrala schemaläggning arkitektur). Analysen visade att belastningen ansökan baseras på distribuerad schemaläggning arkitektur är effektiv, använder mindre resurser testsystem, och kan skala upp kapaciteten last generation.

Nyckelord: FSM, schemaläggning, systemtest, TITANSim, testa komponenter

Dedicated to my parents

Abiramasundari Ramalingam and Kalaichelvan Visagan

Acknowledgements

I express my deepest gratitude to Prof. Gerald Q. "Chip" Maguire Jr., KTH for his valuable feedback throughout this thesis project.

My sincere thanks to Mr. Johan Blom at Ericsson AB for providing this thesis opportunity. Thanks to Mr. Fredrik Berntsson at Ericsson AB for his managerial support throughout this project.

I express my gratitude to Mr. Peter Dimitrov, industrial supervisor, at Ericsson AB for his technical support and feedback throughout this project.

My sincere thanks to Mr. Tao Huang, Mr. Magnus Nilsson at Ericsson AB for helping me to understand the BAT and clarifying many of my doubts related to performing the system test of the HSS.

I thank the TSP Coordination and Management (TCM) team at Ericsson in Spain for their timely support in TSP and HSS installation and solving other environment issues.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

3GPP	3rd Generation Partnership Project
AIA	Authentication Information Answer
AIR	Authentication Information Request
API	Application Programming Interface
AS	Application Server
ASN.1	Abstract Syntax Notation One
AuC	Authentication Centre
AV	Authentication Vector
AVG	Authentication Vector Generator
AVP	Attribute Value Pair
ATS	Abstract test suite
BER	Basic Encoding Rules
BSF	Bootstrapping Server Function
CD	CODEC
CEA	Capabilities Exchange Answer
CER	Capabilities Exchange Request
CH	Component Handling
CLA	Cancel Location Answer
CLL	Core Load Library
CLR	Cancel Location Request
CN	Core Network
CODEC	Coder/Decoder
CORBA	Common Object Request Broker Architecture
CPS	Calls per second
CPU	Central Processing Unit
CS	Central Scheduling
CSCF	Call Server Switching Function
DDF	Data Definitions File
DPMG	Diameter Protocol Module Generator
DS	Distributed Scheduling
DWA	Device Watchdog Answer
DWR	Device Watchdog Request
DWT	Device Watchdog Timer
EDS	Encoding/Decoding System
EPS	Evolved Packet System
EPTF	Ericsson Performance Test Framework
ESM	EPS Subscription Manager
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
ExecCtrl	Execution Control
FBQ	Free-Busy Queue
FSM	Finite State Machines
GPRS	General Packet Radio Service
GSM	Global Services Mobile
GSN	GPRS Support Node
GUI	Graphical User Interface
HC	Host Controller

HLR	Home Location Register
HPLMN	Home Public Land Mobile Network
HSS	Home Subscriber Server
HTTP	Hyper Text Transport Protocol
I-CSCF	Interrogating – Call Server Switching Function
IDA	Insert Subscriber Data Answer
IDE	Integrated Development Environment
IDR	Insert Subscriber Data Request
IMS	IP Multimedia Subsystem
IP	Internet Protocol
IPC	Inter Processor Communication
IPL2	IP Layer 2
IPL4	IP Layer 4
ISM	IP Subscription Manager
LAN L2	Local Area Network Level 2
LDAP	Lightweight Directory Access Protocol
LGen	Load Generator
LM	Load Module
LSV	Latest System Version
LTE	Long Term Evolution
MAA	Multimedia-Auth-Answer
MAR	Multimedia-Auth-Request
MC	Main Controller
MGCF	Media Gateway Control Function
MME	Mobility Management Entity
MSC	Message Sequence Chart
MTC	Main Test Component
NM	Node Management
NSP	Network Server Platform
O&M	Operation and Maintenance
OS	operating system
P-CSCF	Proxy – Call Server Switching Function
PA	Platform Adaptor
PPA	Push-Profile-Answer
PPR	Push-Profile-Request
PTC	Parallel Test Component
RTA	Register-Termination-Answer
RTR	Register-Termination-Request
S-CSCF	Serving – Call Server Switching Function
SA	SUT Adaptor
SAA	Service-Assignment-Answer
SAR	Server-Assignment-Request
SCF	Service Control Function
SCTP	Session Control Transmission Protocol
SDA	Subscriber Data Access
SDK	Software Development Kit
SIP	Session initiation protocol
SLF	Subscriber Location Function
SNMP	Simple Network Management Protocol
SS7	Signaling System 7

SSL	Secure Sockets Layer
SUT	System under test
T3RTS	TTCN-3 Runtime System
TCI	TTCN-3 Control Interface
TCP	Transmission Control Protocol
TE	TTCN-3 Executable
TL	Test Logging
TM	Test Management
TMC	Test Management and Control
TP	Test Port
TRI	TTCN-3 Runtime Interface
TSP	Telecom Server Platform
TTCN	Testing and Test Control Notation
UAA	User-Authorization-Answer
UAR	User-Authorization-Request
UDP	User Datagram Protocol
UE	User Equipment
ULA	Update Location Answer
ULR	Update Location Request
USSD	Unstructured Supplementary Service Data
VoD	Video on Demand
VPLMN	Visited Public Land Mobile Network
WiMAX	Worldwide Interoperability for Microwave Access
WSM	Wireless Subscription Module

1 Introduction

This chapter will introduce the problem area of the thesis project and provide a review of the existing system test architecture and performance, specifically the Home Subscriber Server (HSS) node used in networks such as IP Multimedia Subsystem (IMS) and Long Term Evolution (LTE). The following chapter gives an overview of Testing and Test Control Notation-3 (TTCN-3) programming language features and the test framework used for this thesis project.

System test is a very crucial phase in the development cycle of any software product, since it is generally performed after integration testing and verifies the conformance of product to its stated requirements. For telecommunication (telecom) vendors (such as Ericsson, Nokia Siemens Networks, and Alcatel-Lucent) that are involved in the development of core network nodes in IMS, 3G, and LTE, the system test not only verifies the compliance of the product with *functional* requirements, but other crucial system aspects such as system capacity, call quality, correct generation of operation and maintenance events (e.g., alarms and logging), load balancing, and load regulation are tested.

For telecom vendors the system test phase is very expensive, because the resources (technical labor, time, and computing power) involved in this phase are huge. The system under test (SUT) is installed on telecom grade hardware and the nodes that interact with the SUT are emulated during the system test. Typically the hardware is a clustered multi-processor environment that can handle millions of calls per second (CPS), with huge databases that store customer and call/session related information. The system test strives to reproduce a real field deployment of the SUT. While this is inevitably expensive, it is less expensive than trying to debug a system once it has actually been deployed in the field.

Software testing technologies have improved by leaps and bounds in the last decade, thanks to the efforts of many researchers (in academia and industry), corporations selling testing tools, standards organizations, and the software vendors themselves who have invested in research on software testing. Today a great deal of software testing is automated. This has resulted in a significant reduction in the time and effort needed for development of a test framework, test cases, and the test execution.

TTCN is one such success story – it is a result of the combined effort of the European standards body ETSI (European Telecommunications Standards Institute), leading telecom vendors (such as Ericsson, Nokia, Motorola, and many others), and researchers at some of the finest technical universities of Europe [1]. TTCN-3 (version 3) as it is known today, is an internationally standardized programming language for testing. TTCN-3 is generic and has been successfully employed for testing in telecommunication, automotive, and various other domains of software testing for over a decade [2]. TTCN-3 is the prime driver of automation in telecom testing, since it is used by Ericsson, Nokia, and other major telecommunications vendors for various levels of testing, including functional, system, and performance testing.

Its usage has lead to a significant reduction in software maintenance costs and reduction in the lead time of product development, hence its wide spread usage in the industry [3].

A number of tools that offer a TTCN-3 compiler and an Integrated Development Environment (IDE) are available under open source license, such as BBT and Trex, to name just two [4]. Additionally, Ericsson has developed its own customized TTCN-3 tool called TITAN which is widely used in product development units across Ericsson [5]. TITAN supports numerous protocols (such as Diameter, SIP, DNS, and LDAP) and offers test ports (for several transport protocols, including TCP and SCTP). It is available as a plug-in to the Eclipse IDE* and has its own log viewer [3].

The TITAN test tool was further enhanced to support simulated loads, i.e. to act as traffic generators for performing load testing on a SUT. Within Ericsson, the load capable version of the TITAN tool is commonly referred to as 'TITANSim'. TITANSim provides a very efficient test design and test execution environment for system testers. It has a layered architecture consisting of a TTCN-3 test executor, test ports, protocol support, libraries, and ready to use load applications [6]. A detailed architectural overview of TITANSim is provided in Chapter 4.

This thesis project has taken place within the system test team at Ericsson AB which develops load test applications for the HSS node in Ericsson's implementation of IMS. Within the IMS, the HSS stores the subscription information for each subscriber, performs authentication and authorization of users, and can provide information to others about the subscriber's location and current IP address [7]. HSS is also used in Long Term Evolution (LTE)/4G networks for the same purpose. As per the **3rd Generation Partnership Project** (3GPP) IMS architecture, the HSS communicates over Diameter interfaces with other IMS network elements, such as the Subscriber Location Function (SLF), Call Server Switching Function (CSCF), and Application Server (AS).

During a typical system test of a HSS (i.e., where the HSS is the SUT), the HSS application is installed on the target hardware and the other network elements that communicate with the HSS are simulated by the TITANSim. These simulated components are called traffic or load generators, since their objective is to generate a configured amount of traffic in order to put the HSS under a load for testing purposes. The TITANSim framework which is used to create these load test applications for HSS supports two kinds of architectures: Central Scheduling (CS) and Distributed Scheduling (DS). The current load application is based on using the central scheduling architecture of TITANSim, thus several thousand load generator test components are required to simulate the load when the HSS is subjected to a high target load, for instance 2000 CPS. On the operating system level each of these load generator components executes as a separate process. With such a large number of separate processes, memory consumption is huge when simulating high load conditions. This also results in very inefficient test execution, with slow system response times. As the current load test application is based on using the centralized scheduling architecture of TITANSim, load

* Eclipse IDE, www.eclipse.org

testing consumes a lot of memory and this approach is inefficient when simulating high load conditions, both factors increase the cost of the system test process [6]. In the central scheduling architecture based load test application, the scheduling of test components is centralized which results in a lot of communication overhead within the test system, as thousands of test components communicate with a *master scheduling* component during the test execution.

This thesis project aims to create a load test application based on the distributed scheduling architecture of TITANSim, thus multiple traffic cases will be simulated using a single test component. This will greatly reduce the number of operating system level processes involved, thus greatly reducing memory consumption. This approach will enable high load conditions to easily be simulated with limited hardware resources. In the distributed scheduling architecture the scheduling task is performed *locally* by each test component. There is no communication overhead within the test system. Therefore, the test system is highly efficient. In the distributed scheduling architecture based load test application the traffic flow of the simulated users are described using the Finite State Machines (FSMs). The FSMs are specified in the configuration files that are used by the test system at run time. Therefore, implementing traffic cases using the distributed scheduling architecture becomes simpler and faster as there is no (TTCN-3) coding/compilation.

2 The SUT - HSS

In the real world, system test developers require a deep understanding of the SUT and the protocols involved, because a system test is performed on real target nodes. Hence, the system testers have to install, configure, and provision the SUT software on the target machine in order to perform the system test.

The SUT for our thesis project is an HSS, which is the master database node in IMS, 3G, or LTE networks. In the following discussion we will focus on its role and operation in IMS (later we will consider the role of the HSS in mobility management). Architecturally IMS consists of a transport layer, IMS layer, and a service/application layer. The basic philosophy of IMS is to be access-agnostic, so that users from any kind of access network can connect to an IMS enabled service. Hence, the transport layer consists of a number of gateway nodes which allow devices from a variety of access networks (such as WLAN, 3G/LTE, WCDMA, PSTN, and WiMAX) to access the IMS. The IMS layer consists of core switching function nodes such as S-CSCF, P-CSCF, and I-CSCF. These types of nodes form the heart of IMS and perform the important function of call/session handling. Note that IMS is based on the use of the session initiation protocol (SIP) [16].

The HSS is present in the service/application layer of IMS, along with a number of application servers (AS). An AS can be used to provide a variety of services to the end users, such as video on demand, video-conferencing, Push-2-Talk, and so on. End users who access these services using the IMS network must have an IMS subscription. In addition, each subscriber can have a profile which stores information related to the subscriber's service preferences, authentication information, and so on. All IMS subscriber related information is stored in the HSS. The HSS performs the authorization and authentication of the user along with providing location information and the IP address information of the subscriber's UE to other nodes upon request. A typical home network in IMS will have millions of subscribers; hence the subscriber data will be spread over multiple HSSs. Therefore as noted in section 3.2.2, a SLF is used to identify in which HSS a particular subscriber's data is stored. Figure 1 shows an HSS interacting with other IMS nodes through Diameter interfaces Cx and Dx [12]. The HSS is also used in LTE networks, where it interacts with the Mobility Management Entity (MME) through the S6a Diameter interface [13]. The module within the HSS that manages the S6a interface for LTE access is called the Enhanced Packet Services (EPS) Subscription Manager (ESM).

Figure 2 shows the EPS architecture and its interfaces. In this thesis we will restrict our discussion to the S6a interface. The S6a interface enables the transfer of subscriber related data between a MME and HSS. The MME is the heart of the mobility support in an LTE access network, as it tracks the UE and is responsible for handover when the UE moves from one node to another node within a given LTE network[†].

[†] Actually an MME can also handle handovers to/from non-LTE networks, but these details lie outside the scope of this thesis.

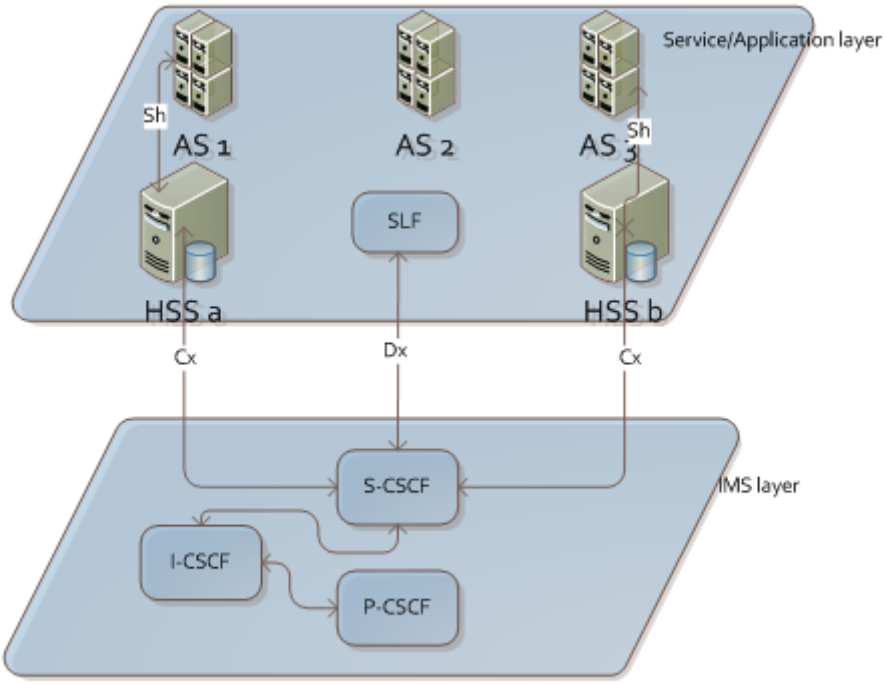


Figure 1: IMS Application and Control layer

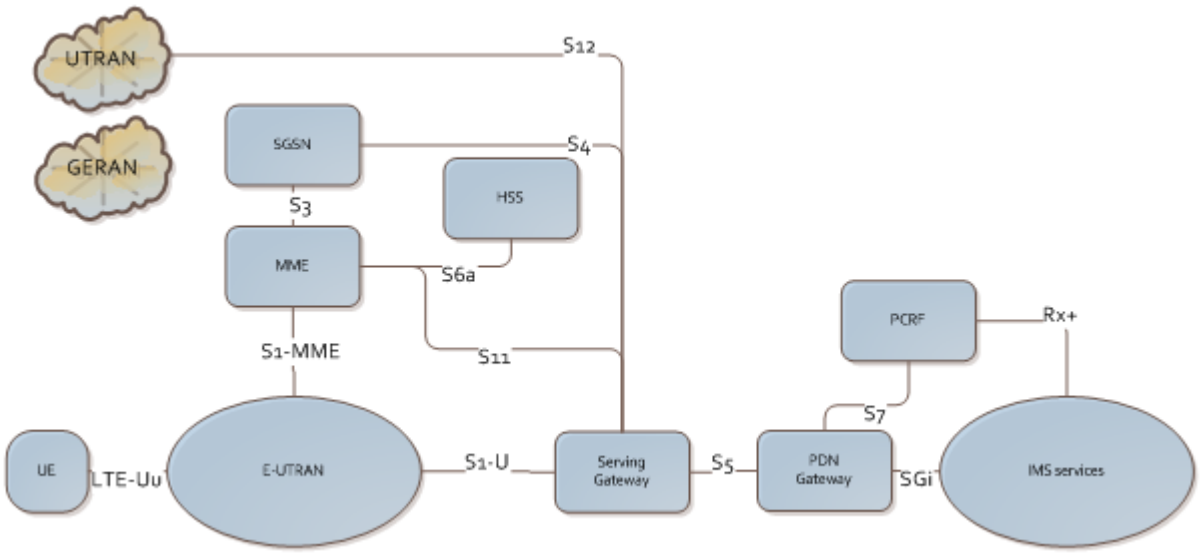


Figure 2: EPS Architecture

The update location procedure is used between the HSS and MME to update the location information of the subscriber in the HSS database. This is done by sending the Diameter S6a messages: Update Location Request (ULR) and Update Location Answer (ULA). In addition to location information, other information such as the MME's identity and terminal information are exchanged with the HSS in the Update Location procedure. When a

subscriber's subscription has terminated the HSS uses the Cancel Location procedure to inform the MME, using the Diameter S6a messages Cancel Location Request (CLR) and Cancel Location Answer (CLA). When any of the subscription related data is changed in the HSS, the MME is informed by the HSS using an Insert Subscriber Data procedure via the Insert Subscriber Data Request (IDR) and Insert Subscriber Data Answer (IDA). The MME can also retrieve authentication information for a subscriber from the HSS through the Authentication Information Request (AIR) and Authentication Information Answer (AIA) messages. Figure shows the S6a interface with some of these messages.

For this thesis project the S6a interface between the MME and HSS has been chosen for developing the load application based on a distributed TITANSim load scheduler. The S6a interface has been chosen because from a TITANSim perspective, the LGen design is less complex and the required number of traffic generators needed is less than directly using the Cx or Sh interface. From the HSS's (i.e., the SUT's) perspective, the configuration and provisioning of a HSS is simpler, as the S6a interface requires fewer modules within the HSS. Thus the S6a interface fits well within the scope of the thesis work with regard to the time available to carry out this project and the complexity of problem that can be adequately addressed in this period of time.

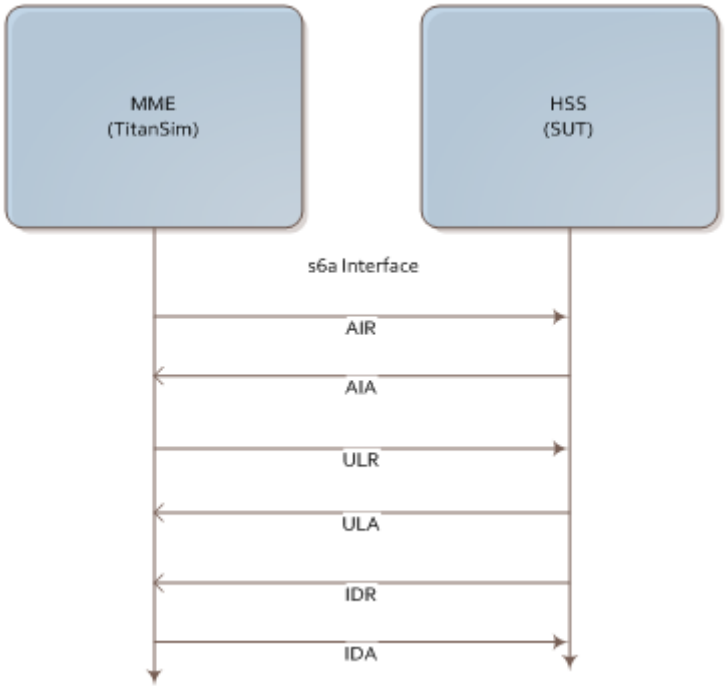


Figure 3: MME – HSS (S6a interface)

3 Background: TTCN-3

Since the mid 1980s, TTCN-3 has become the *de facto* standard for testing and writing test specifications. It is an internationally standardized language and offers tool vendor independence. Therefore any TTCN-3 tool will compile the test scripts in exactly the same way. It is a simple high level programming language suitable for test developers to create test specifications, even for complex systems with multiple protocols/interfaces. It has a textual format for defining test cases – referred to as the TTCN-3 core notation. It offers many presentation formats for specifying test cases ([8], page 18). Test cases can be specified in tabular format, Message Sequence Chart (MSC) format, or in a simple textual format. All of these presentation formats can be converted to the TTCN-3 core notation. TTCN-3 supports different communication modes, such as message oriented communication and procedure based communication and also includes numerous features (such as built in data matching, concurrent test execution, timers, and a distributed test architecture). The following sections provide a brief architectural overview of a TTCN-3 test system and describe some of its core language features.

3.1 TTCN-3 test system

A test specification consisting of several test cases written in the TTCN-3 core notation constitutes a test suite. A TTCN-3 test suite is abstract, commonly referred to as an abstract test suite (ATS), since it is devoid of any system specific information, such as choice of Coder/Decoder (CODEC) or transport protocol ([9], page 22). This abstract nature of the test suite makes it easier for the TTCN-3 test developers, as they only need to focus on the *functional* aspects of the test (such as specified by a protocol message sequence or finite state machines) when creating test suites. A TTCN-3 test system is comprised of several other parts in addition to the test specification itself. These other parts are the TTCN-3 Executable (TE), SUT Adaptor (SA), Platform Adaptor (PA), and Test Management and Control (TMC) ([9], page 23).

The above functions are realized by several test system entities that interact with each other through standardized interfaces: TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI). Figure 4 shows the different entities involved in the TTCN-3 test system. The core of the TTCN-3 test system is the TE which executes the TTCN-3 statements. The TTCN-3 execution layer communicates with the PA and SA though TRI and with the TMC layer through TCI. ETSI has standardized these interfaces in [10] and [11].

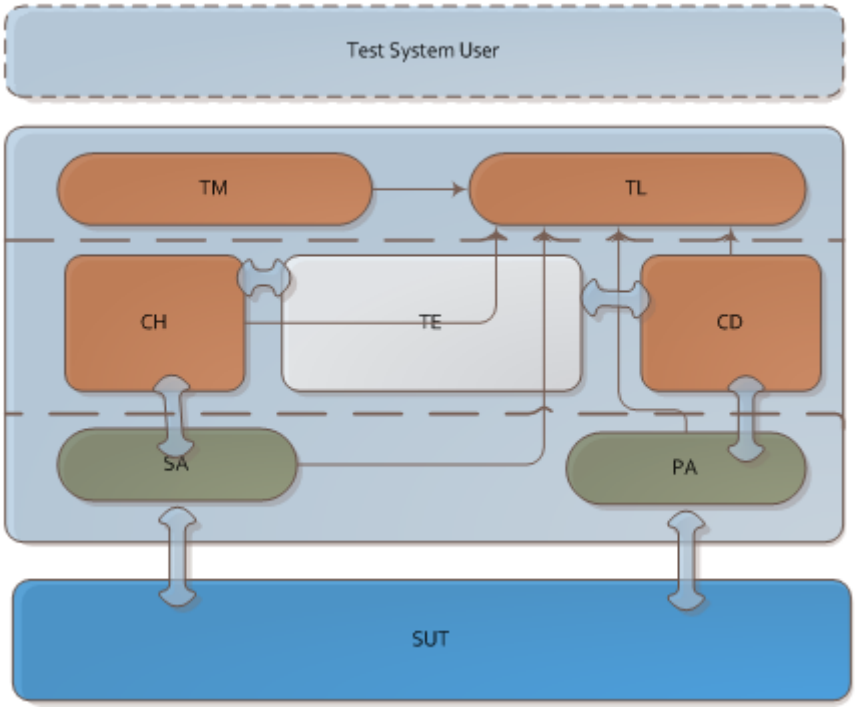


Figure 4: A TTCN-3 Test System[‡]

3.1.1 Test Management and Control

The test management and control (TMC) layer internally consists of three entities: CODEC (CD), Test Management (TM), and Component Handling (CH); assisted by a Test Logging (TL) entity.

CD The CD entity performs the encoding and decoding of messages exchanged between the TTCN-3 test system and the SUT, so these messages are understandable to each system. All TTCN-3 tools support standardized encoding schemes, such as ASN.1 [32], and BER [33]. However, if the SUT uses proprietary encoding schemes, then a user defined CODEC should be used.

TM The TM entity helps a test developer handle the test process itself. The test developer can control the order of execution of test suites or block some part of the test suite from execution. This control enables the tester to customize the test process to suit his or her needs. Log creation, viewing, and debugging are simplified by using a Test Logging (TL) entity.

CH The CH entity plays a key role in distributed test execution. In a distributed test execution, the test executable (TE) is spread across multiple test nodes. A typical distributed TTCN-3 test system is shown in Figure 5.

[‡] Adapted from Figure.1 General Structure of TTCN-3 Test System, Page 12 of [10]

TL The TL plays the key role of logging of the test events and making these events available to the test user. Typical test events include send/receive messages between the test system and SUT, alarms, timers, component creation, execution, and termination information.

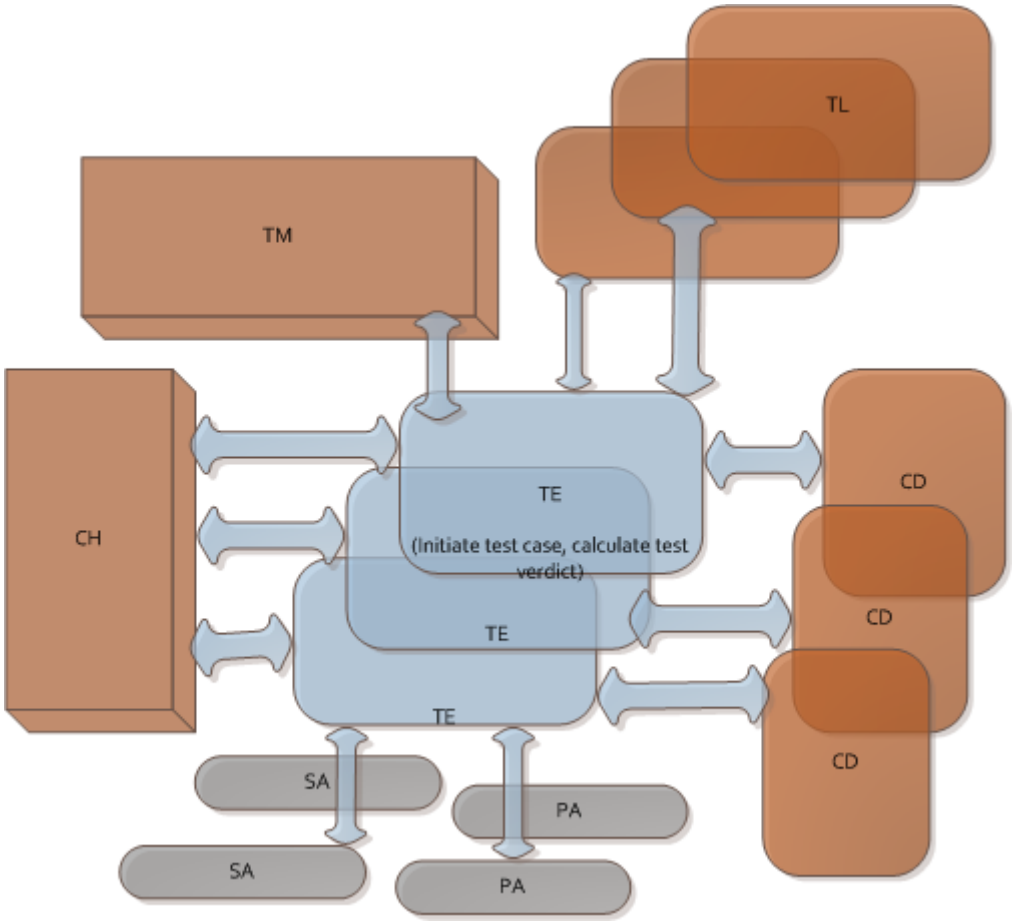


Figure 5: A Distributed TTCN-3 Test System[§]

Each node involved in the distributed test system will have its own SA, PA, CD, and TL entities. The CH and TM entities manage the communication between test entities in each of the nodes in the distributed test setup. The CH entity is aware of all of the communication ports of all the test entities in all nodes, thus the CH can forward messages from one instance of an entity to another instance of the same entity in a different node.

3.1.2 TTCN-3 Executable (TE)

The TE entity is responsible for the execution of a specific TTCN-3 test suite (the ATS). The TE is conceptually divided into an Executable Test Suite (ETS), a TTCN-3 Runtime System (T3RTS), and an optional Encoding/Decoding System (EDS) ([10], page 13).

[§] Adapted from Figure 3 , Page 21, [11]

ETS The ETS entity executes the test cases. During the test case execution, ETS communicates with the T3RTS entity to send messages to the SUT. When the response message is received the ETS entity matches the received response with the test case defined template of expected response as encoded in the test suite.

T3RTS T3RTS is the heart of the TE entity. It communicates with the TM entity through the TCI interface and with PA and SA through the TRI interface. All the test events are logged by the TM entity with the help of T3RTS. The T3RTS is responsible for initializing the adaptors, i.e., the ETS and EDS entities. T3RTS notifies the SA entity what message is to be sent to the SUT. It communicates with the PA entity so that appropriate timers are started, stopped, read, or simply queried during the test execution. T3RTS invokes the EDS entity so that all the messages are appropriately encoded / decoded before they are exchanged with the SA.

EDS The EDS entity is responsible for encoding and decoding of all test data exchanged between the T3RTS and SA during the test execution.

3.1.2.1 SUT Adaptor (SA)

The TTCN-3 test suite is completely abstract. Hence, the mapping of all TTCN-3 operations to a real world operation is performed by the SUT Adaptor (SA). For example, while a test suite specification might simply indicate the message to be sent through a port, the SA entity maps the TTCN-3 port to a real world port (i.e., to a specific destination IP address, transport protocol, and port number).

3.1.2.2 Platform Adaptor (PA)

The platform adaptor (PA) entity handles all the events that occur when messages are lost between the TTCN-3 test system and the SUT. For example, timers need to be implemented to handle message timeouts that arise when messages are lost in transport or the communicating node fails to respond to a specific message. Timers are implemented at the platform level (with messages such as Device Watchdog Timer (DWT) that monitor connectivity with a Diameter peer node), and also at the application level as per the specific protocol's specification.

3.1.3 TTCN-3 Control Interface (TCI)

In a test system all the communication between the TE, TM, CH, CD, and TL entities are defined by the TCI. TCI enables a TE to manage test execution, log events with TL, distribute and coordinate test entities in different test devices, and perform coding and decoding ([11], page 22).

3.1.4 TTCN-3 Runtime Interface (TRI)

TRI defines the communication between the TE & PA and TE & SA. TRI enables a TE to exchange messages with the SUT through SA, control timers, and receive messages and

timeout events. The ETSI standardized interfaces TRI and TCI enable a TTCN-3 test system to be adapted for any kind of SUT ([10], page 15).

3.2 TTCN-3 Core Language Features

ETSI has designed TTCN-3 to be a very flexible and powerful language for performing system tests ([8], page 17). The language features that are built into TTCN-3 are extensive and make TTCN-3 suitable for regression testing, integration testing, and stress testing. However, there are some specific TTCN-3 features that are pertinent to the development of TITANSim. In this section we will briefly discuss those features that make TTCN-3 very powerful for load test application development.

3.2.1 TTCN-3 component architecture

Any load test application designed in TTCN-3 is built on a number of components. Depending on the complexity of the testing task, a single component test system can be designed (with only a Main Test Component - MTC) or the system can involve multiple components (Parallel Test Components - PTCs) – as shown in Figure 6. In test systems that involve multiple components, the MTC is responsible for creating the PTCs. Each PTC will perform its defined testing task and produce its own component level test verdict. The MTC collects the verdicts from each of the PTCs and determines the final verdict for each particular test case.

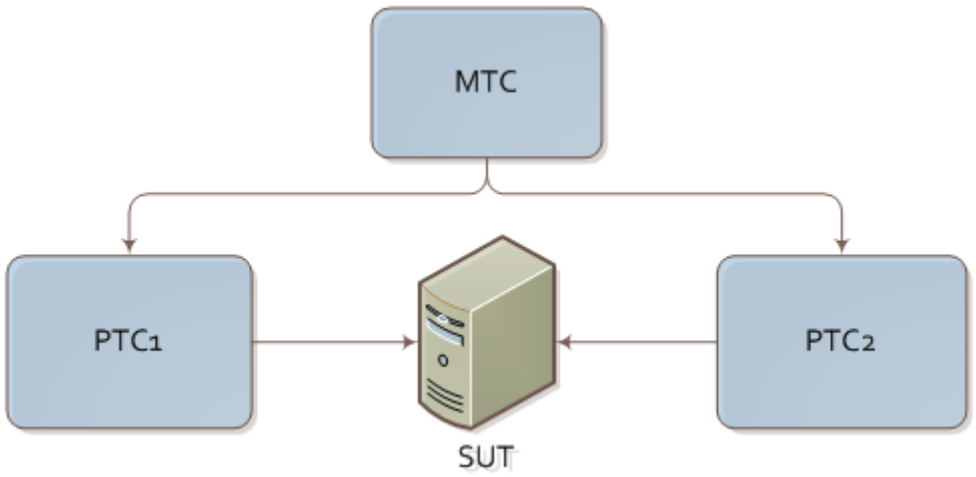


Figure 6: SUT interacting with various test components

Each component is defined with a particular type name and its definition includes associated ports, timers, variables, and constants of the component. As will be discussed in section 3.2.3 TTCN-3 components communicate with each other and with the SUT using ports. The component type definition is local to a particular component hence other components may use the same port names or timer names in its definition. The following code sample shows a typical component definition with port, timer, and a variable.

```
type component MyPTC1Type
{
var integer MyLocalInteger;
timer MyLocalTimer;
port MyMessagePortType PCO1
}
```

It is also possible to define a new component type by extending an already defined component type using **extends** keyword; the former is referred to as the extended type and the later as the parent type. It is also possible to have an extended type component definition extending multiple parent types. In such a case the parent type definition may also be based on an extension. So effectively the component definition of an extended type includes all the ports, variables, timers defined in the parent type as well as those defined directly in the extended type.

```
type component MyMTCType
{
var integer MyLocalInteger;
timer MyLocalTimer;
port MyMessagePortType PCO1
}
type component MyExtendedMTCType extends MyMTCType
{
var float MyLocalFloat;
timer MyOtherLocalTimer;
port MyMessagePortType PCO2;
}
```

3.2.2 Concurrency

In any telecommunication network (such as IMS, GSM, or 3G/LTE) the SUT is quite complex as it communicates with multiple nodes via many different interfaces. For example, in the IMS network, the HSS node contains the subscriber database storing all the subscription information needed in order to perform authorization and authentication of users. Typically, the number of subscribers is very large. Hence, the home domains in actual IMS networks have multiple HSS nodes. In this case, the IMS systems needs a Subscriber Location Function (SLF) in the home domain to map the user's IMS address to the corresponding HSS that stores this particular user's profile and other subscription information. The SLF is used by a number of nodes in the IMS home domain, specifically the Serving – Call Server Switching Function (S-CSCF). When the S-CSCF wants to retrieve any information from a user's profile or wants to authenticate a user using the HSS database, the S-CSCF must first locate the SLF and ask it for the address of the appropriate HSS for this specific user. Then the S-CSCF establishes a session with the appropriate HSS (unless it already has an existing session). The HSS provides the user with access to Application Servers (AS), thus the HSS facilitates authentication of a subscriber using a particular device (i.e., User Equipment (UE)), when this UE wants to initiate a session with the AS. To realize

a secure and authenticated connection between the UE and AS, the HSS uses another node called the Bootstrapping Server Function (BSF) that provides security keys for the session between UE and AS. Thus a simple scenario such as retrieving a user profile or performing user authentication will involve multiple nodes. Figure 7 shows the S-CSCF, SLF, and HSS along with the Diameter interfaces that they use to communicate.

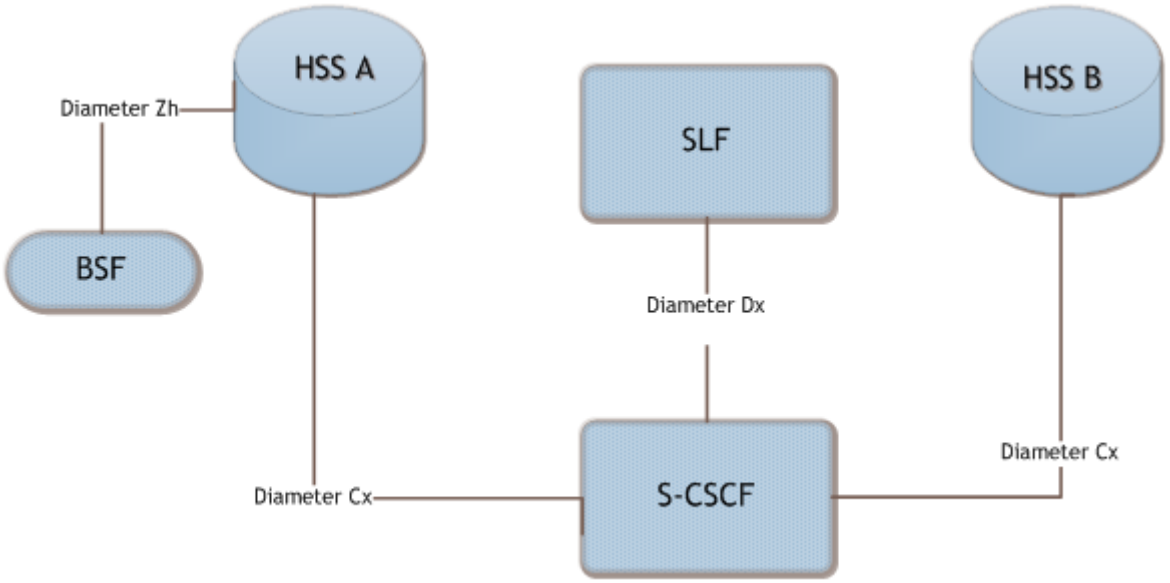


Figure 7: HSS as a SUT interacting through multiple Diameter interfaces

If we consider HSS A as the SUT, our test system should have at least two test components, one each for Zh and Cx interfaces. Thus one test component (say PTC1) would simulate a BSF node attached via the Zh interface and the other (say PTC2) would simulate S-CSCF node attached via the Cx interface. It is also possible to design the same test system with a single component, but this version will operate in a non-concurrent fashion (e.g., first simulating interactions via the Zh interface, then Cx or vice versa). However, this non-concurrent design is very complicated and the test system code will be hard to maintain. In contrast, a concurrent TTCN-3, design is modular, hence it is easier to maintain – even for a system as complex as shown in Figure 7. For concurrent TTCN-3 design it is important to have a very clear definition of the configuration of the test components. The test component configuration includes the details of the ports with which the test components communicate with each other and test system interface with the SUT.

As with any TTCN-3 test system, a concurrent TTCN-3 test system includes one MTC, which is responsible for the creation of the PTCs. The TTCN-3 core language provides a special **create** operation to dynamically create PTCs at run time, i.e. during test execution ([8], page 93). Figure 8 shows a typical TTCN-3 concurrent test system configuration.

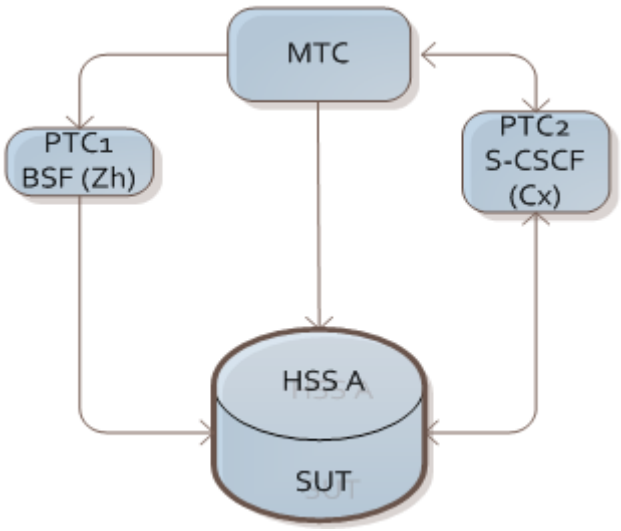


Figure 8: A Concurrent TTCN-3 test system

3.2.3 Synchronous and Asynchronous communication

Each component has a certain number of assigned ports through which it will exchange messages with the SUT. The message types that are part of the definition of a port type can be bidirectional (inout), send only (out), or receive only (in). TTCN-3 port types can be message based or procedure based ports, depending on the communication mode used by the test components. In both the cases the port will have a set of predefined message signatures or procedures that define what the port is allowed to send or receive. The following code samples show the definition of message based and procedure based ports.

```
type port MyMessagePortType message
{
in MsgType1, MsgType2;
out MsgType3;
inout integer
}
type port MyProcedurePortType procedure
{
out Proc1, Proc2, Proc3
}
```

In message based communication mode, as shown in Figure 9, the **send** operation in the sending component is non-blocking, i.e. it is asynchronous, while the **receive** operation in the receiving component is blocking ([8], page 166). For example, a send or receive operation on a message based port is performed as below.

```
MyMessagePort.send(5); // Sends the integer value 5
MyMessagePort.receive; // Removes the first value from MyMessagePort.
```

In a typical real-time system, there will be multiple incoming messages queuing up at the receiving component. To handle such message queues, TTCN-3 provides a **trigger** operation that allows a receiving node to apply some filter conditions on the message queue and to

process only those messages that match the filter criteria. A basic example of such trigger code is shown below. It specifies that the operation will trigger on the reception of the first message of the type MyType with an arbitrary value at the port MyMessagePort.

*MyMessagePort.**trigger**(MyType:?);*

Figure 9: Asynchronous send and receive operation

The procedure based communication mode, as shown in Figure 10, is based on remote procedure calls. The sender node invokes a call operation on the receiving node, for which the receiving node responds by invoking a **getcall** operation. If the operation is successful, then the receiving node will respond with a **reply** operation or raise **exception** operation. On the sender node the response is received using a **catch** operation ([8], page 166). Procedure based communication can be blocking, where both the sender and receiver nodes wait until the operation is performed; or the communication can be non-blocking. The non-blocking mode of communication as shown in Figure 11, where the operation of the sender node is non-blocking, while only the receiving node operates with a blocking state. For detailed handling of procedure based communication with code examples refer to ([8], pages 174-184).

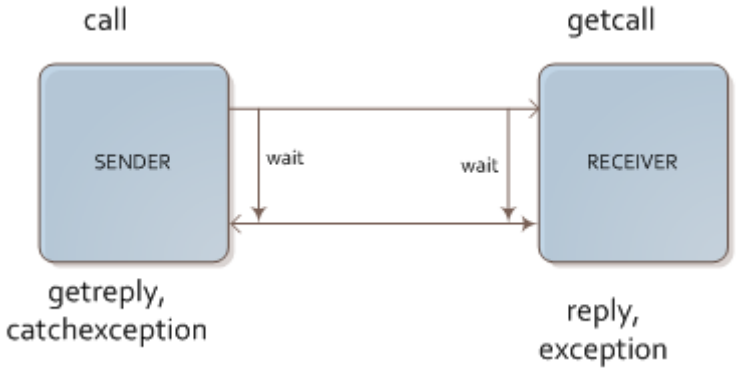


Figure 10: Blocking procedure based communication mode

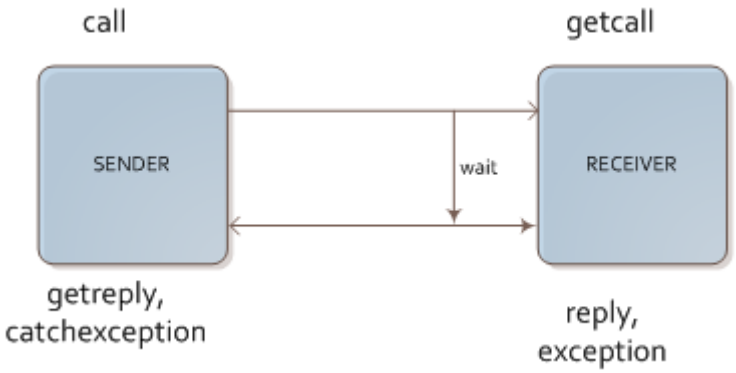


Figure 11: Non-blocking procedure based communication mode

At the TTCN-3 language level, each component is always defined with its associated ports. Thus the component type is differentiated based on the associated ports. The port names are also local to the component, hence they are not unique. The component type definition also includes constants, variables, timers, and functions. Thus every instance of the component type that is created will have its own set of ports, timers and functions as defined in the component type definition.

3.2.4 Timer Handling

As discussed in the section 3.2.2, in non-concurrent TTCN-3 test systems, the execution flow stops till a response is received from the SUT. Hence, the test system should be prepared to handle situations when there is no response from the SUT. TTCN-3 provides timers that expire and indicate to the test system the SUT's inactivity. TTCN-3 provides all the operations needed for creating a timer, checking for an expired duration, stopping a timer, check a timer's status, and blocking execution while the timer is running – as shown in Table 1. Timers are also used to create a sufficient time gap between executions of test cases to enable the tests cases to execute from a stable state. Typically timers are defined in the components' definitions. TTCN-3 also allows timers to be defined inside the individual test cases and in the module's control part. The sample code below shows the different operations that can be performed with a timer.

timer *sampleTimer; //* declare a timer variable

*sampleTimer.**start**;*

*sampleTimer.**stop**;*

var float *timerValue;*

*timerValue := sampleTimer.**read**; //* read elapsed time

if*(sampleTimer.**running**) {} //* check if the timer is running

*sampleTimer.**timeout** //* check the expiration of the timer

Table 1: TTCN-3 Timer operations

Statement	TTCN-3 operation
Start timer	Start
Stop timer	Stop
Read elapsed time	Read
Check timer status	Running
Timeout event	Timeout

For efficient timer handling each component or control module where the timer is defined maintains a list of running timers and timeout timers. Whenever a timer is started an entry is made in the running timers list. When the timer expires, it is added to the list of timeout timers. Thus at any point of time during test execution an entry for each timer exists either in the running timers list or the timeout timers list ([8], page 189). When the execution of the control module or the test component is stopped, all the running timers are cancelled and no longer exist in either list of timers.

3.3 TTCN-3 and finer aspects of system test

In this section, some key aspects of the TTCN-3 based test systems (such as concurrency, resource optimization, and resource management) are discussed.

3.3.1 System testing

In section 3.2.2 about concurrent TTCN-3 testing, a typical example of session authentication using the HSS was discussed. It was quite clear from this example that the TTCN-3 test system has to simulate all the other nodes that interact with the HSS (SUT). In a real world scenario, the HSS in an IMS network may need to authenticate thousands of user sessions per second. In order to apply the same load to the SUT, the test has to simulate tens of thousands of users; however, this will require a lot from the test system – especially if each user is to be individually modeled. This would require that there be tens of thousands of test components involved in the test system, all executing in a concurrent fashion.

Using this approach, the TTCN-3 test system designed to system test a node such as a HSS will be as complex as the HSS itself (or perhaps even more complex). However, the complexity of the test system can be reduced to a certain extent by clearly defining the design scope of the system test. Consider that the process that generates thousands of individual requests for session authentication from the SUT per second does not need to wait for the response from the SUT, but rather this process can execute thousands of concurrent threads where each thread will process the response to its request when the response later arrives. This thread will then take the appropriate action based on the traffic flow that was defined in the test system.

This means that we can test the processing of the SUT in a system test *without* modeling each individual user. The test system developer can design a TTCN-3 test system so that a certain number of authentication requests are made per second to the SUT with a complete set of messages as per the standard protocol specifications**.** Note that only some of the responses actually needed to be checked to see if they are the correct and expected response, the rest can simply be ignored – thus reducing the load on the load generator, enabling a given load generator to simulate a much larger number of users making authentication requests per second. However, the SUT must not know which of the authentication requests are being checked and which are not, thus it must actually process all of them correctly.

3.3.2 Resource optimization

As discussed in Chapter 1, performing a system test is a very resource intensive process. Each simulated user consumes a significant amount of memory in the test system. For example, a real HSS might need to support 10 000 simultaneous users, hence the test system will have to simulate at least 10 000 users. If on an average the process simulating each user consumes 100 kilobytes of memory, then a total of 1 gigabytes of memory is required. The traffic rate of the load generator, hence the load on the SUT is proportional to the number of users that can be simulated by the test system. This means that the TTCN-3 test developers have to strike a balance between achieving high loads and the resource consumption required to generate this load.

The resources required of the load generator can be reduced considerably by re-using resources. Assume we need to generate a load of five calls per second (CPS) by the test system in order to test the SUT under this load and that it takes one second for each call to complete, then at the end of five seconds, instead of deploying a sixth user, we can reuse the first user (who is now free) to play the role of the sixth user. Simplistically this would imply that to achieve a traffic rate of five CPS, all we need is just five users; however, this is not quite true as in generally we would like to avoid the delay of setting up a user to enter the queue – hence we might use a number of additional users to enable the load generator to create users that are prepared to place call, then simply feed these prepared users to the SUT at the desired rate. In this slightly more sophisticated method we can actually test the system with a load of five CPS, rather than a load that varies between four and five with some unknown duty cycle (i.e., the fraction of time when the system is actually under a load of five users could vary from a few percent to 100% of the time – depending upon how quickly the load generator can generate a new user state). In TITANSim a common practice while configuring the load application is to make the load generators simulate a number of additional users which is nearly 20% more than the required number of users. The time taken by the users to execute their assigned traffic case and become available again for execution depends on the system response time with the varying load levels. This approach has been proven in research with SUT's such as an HTTP server, put under test with commercial load generators such as HP's LoadRunner[**] ([17], chapter 5). These results show that the average

[**] Commercial software for performance validation from HP, check https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5e8_4000_100

response time of the system increases as more number of users start using the system (i.e., as more load is generated by the test system). Under high load conditions the SUT response can be very slow (well below satisfactory limits) and the users simulated by the test system will have to wait longer for their response. From a test system's perspective, this means that its resources (user components and the associated memory) take longer to complete execution and become available for reuse. To counter this problem TITANSim provides a load regulation feature to regulate the CPS and maintain the SUT in a stable state during test execution. This feature periodically measures the CPU load level on the SUT and calculates a new target CPS value.

3.3.3 Resource management

Since the TTCN-3 test systems used for system test usually simulate thousands of users to achieve high traffic rates, it uses a lot of memory which can become fragmented during test execution. Memory fragmentation will seriously affect the performance of the test system as it reduces the execution speed. So the load applications built using TTCN-3 should have efficient data structures to avoid memory fragmentation. For instance, TITANSim provides a Free-Busy Queue (FBQ) which is used to maintain a list of free and busy slots in the memory ([6], page 4). The FBQ is used for dynamic memory allocation in any load application developed using TITANSim. So when a user process is created by the test system during execution, an element representing a free resource is removed from the free queue and added to the busy queue and vice versa when the user process terminates**.**

3.4 Related work

Test automation with TTCN-3 has been an area of significant research in telecommunications over the past decade. Development of a test framework assumes as much significance as the software itself. Markus Warken in his publication titled '*From Testing to Anti-product Development*' illustrates how telecommunication companies consider modern testing as an anti-product development [24]. Old ways of software development models such as the Waterfall model [25] have now been replaced by parallel software development (also known as incremental development) methods such as Scrum [26], which necessitate the development of the anti-product in parallel to the product. This has also led to a general understanding in the telecommunications industry that the anti-product is in itself a complex software system and that the test development teams must have competent developers. Markus draws onto his vast personal experience in anti-product development using TTCN-3 and explains how TTCN-3 is the method of choice for telecommunications testing. He also explains the advantages of using TTCN-3 for anti-product development such as the concept of re-use in the test system development, and the ease of test management, automation, and debugging ([24], pages 303, 304).

Any anti-product developed for the system test focuses on two main goals. Firstly, the test system should not be as complex as the SUT itself. Secondly, the test system should generate sufficient load to test the load characteristics of the SUT and at the same time use minimal resources. While technologies such as TTCN-3 can be effectively employed to achieve the

first goal of a simpler, easy to maintain test system; the second goal has to be addressed at an architectural level of the test system; independent of the technology used to develop it. Ferenc Bozóki and Tibor Csöndes of Ericsson Hungary in their publication titled '*Scheduling in Performance Test Environment'* provide a Finite State Machine (FSM) based architecture and an algorithm to improve the scheduling efficiency in such test systems [27]. In the functional test, typically every test case simulates just one user (user or client of the SUT) using a POSIX thread (for general information on such threads see [28]). This is called the '*one user-one thread*' approach. Unfortunately, while simple – this approach does not scale up well for a performance test environment where thousands of such threads are required. This would consume a lot of resources and the thread scheduler will spend time switching between the threads during the test execution. To solve this problem, Ferenc and Tibor abstract the concept of a thread to an abstract level called *virtual threads*, where a single thread could emulate multiple users, and the execution of each user is considered as a virtual thread. This is called the '*multiple user-one thread*' approach ([27], page 2). It is realized using FSM theory [29]. The '*multiple user-one thread*' approach also has some bottlenecks. As almost every message exchanged between the test system and the SUT is timer driven, the test system has to process each message within a time; otherwise the message is considered too late leading to a false (test) verdict. Thus, there could be undesirable delays in processing of events in a '*multiple user-one thread*' approach as a single thread is responsible for handling events for thousands of users. To solve this problem, Ferenc and Tibor propose an algorithm to reduce the total delay of the test system by optimizing the scheduling of events ([27], page 4).

In the system test environment, the test system used for generating load typically is comprised of multiple test nodes. When multiple test nodes are involved, test component distribution becomes vital to efficient utilization of test system resources during the test execution. George Din, Sorin Tolea, and Ina Schieferdecker in their publication titled '*Distributed Load Tests with TTCN-3'* provide an architecture for distributed load test execution and also present a few distribution algorithms along with their characteristics [22]. They also present three generic patterns of test component specification, namely a *one component per client* pattern where a component emulates just one client or user, a *sequential repetition of clients per component* pattern where a component sequentially repeats in a loop emulating a different user each time, and an *interleaved client behaviors per component* pattern where a component simulates multiple users in parallel ([22], page 182).

A number of factors have to be considered when choosing an appropriate algorithm for test component distribution, when the test system involves multiple test nodes. For example, in the *one component per client* pattern test components are created, execute a scenario, and terminate for *every* user. In such a case the balancing of test components (a decision on which test node the test component should be created) can be performed during the test execution at the time of test component instantiation. In the case of *interleaved client behaviors per component* pattern test components execute for *longer periods* since they simulate multiple users ([22], page 184). For this pattern the test component distribution needs to be performed before the test execution using the knowledge of the resources consumed by the test

components and how much the relevant resource is available on each test node. Based on when the distribution decision is taken, the distribution algorithms can be classified as static or dynamic ([22], page 189). A static algorithm, such as *round robin test component distribution,* is effective when the test nodes have more or less same resources (for example memory and CPU capacity). It should be noted that the *test component distribution was performed in a similar fashion for the tests in this thesis project* (refer to section 5.5.2). Dynamic algorithms are effective when the test nodes have different memory and CPU capacity ([22], page 190). Dynamic algorithms use a threshold parameter such as memory, or load level. The algorithm periodically measures the threshold parameter on each test node during the test execution and accordingly decides on which test node the test component should be deployed.

In '*Distributed functional and load tests for Web services',* Ina Schieferdecker, George Din, and Dimitrios Apostolidis, implement a distributed test platform architecture for load testing a simple web service [23]. They experiment by increasing the test load to measure the changes in response time of a web server with an increasing number of test components. Such studies are essential for striking the correct balance between SUT load levels, number of test components, and test execution time.

4 TITANSim

Now that we have equipped ourselves with a basic understanding of the TTCN-3 test system architecture, its function and language features, we will now study a real world TTCN-3 system that is used widely within Ericsson for system test load application development.

TITANSim is an extensible, modular, software load test framework written in the TTCN-3 language. It provides all the necessary building blocks to build any load test application. TITANSim utilizes all the core features that were discussed in the previous chapter to their fullest potential. It provides all the functionalities required to develop highly optimized load test applications needed for carrying out a system test. The TITANSim features can be broadly classified as protocol dependant and protocol independent. Protocol *dependant* features include several libraries for protocols such as SIP, Diameter, HTTP, MAP, DNS, test ports, and many more. Protocol *independent* features include data structures needed for resource management such as the FBQ, Hash Map, and user scheduling components.

4.1 TITANSim Architecture

TITANSim is an extensible modular software load test framework written in TTCN-3 language for TITAN, Ericsson's proprietary tool for TTCN-3 test execution. The main architectural elements of TITANSim are the Test Ports (TPs), TITAN libraries, Core Load Library (CLL), Application Libraries, Control Logic, and Run time Graphical User Interface (GUI) ([6], page 6) – as shown in Figure 12.

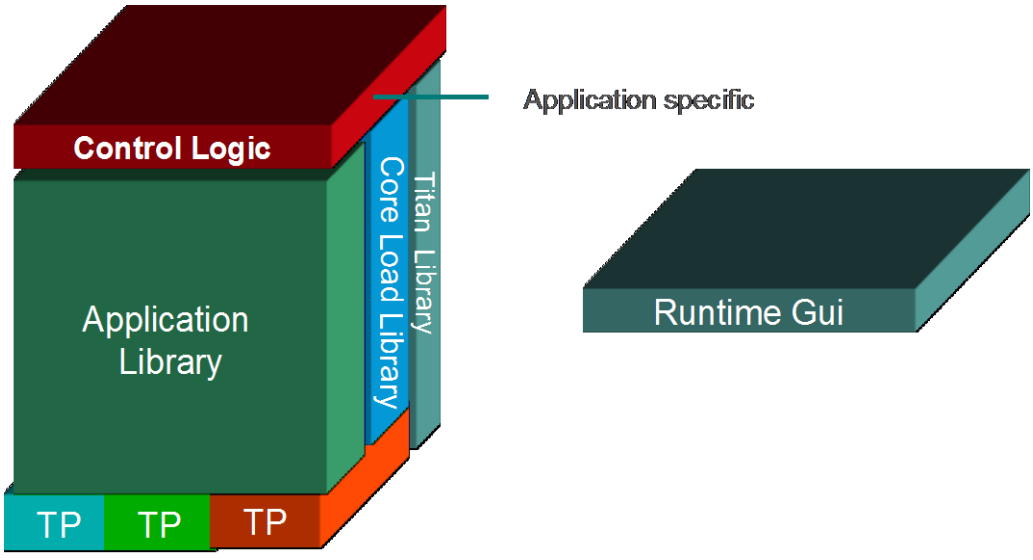


Figure 12: TITANSim architectural elements

TITAN Libraries The TITAN libraries provide all the basic TTCN-3 language functions & operations, such as **send**, **receive**, and **create**. These libraries also provide all the functions needed for communication between TTCN-3 components and the SUT, data structures and data types, and logging.

Test Ports The TPs implement the transport functions for sending and receiving messages. They also perform the CODEC functions for all messages exchanged with the SUT.

Core Load Library The CLL forms the protocol *independent* part of the TITANSim architecture. The CLL provides all functions needed for user creation, resource management functions (such as FBQ and hash map), user scheduling, and load regulation.

Application Libraries Application libraries provide the protocol *dependent* part of the TITANSim architecture. A number of protocol application libraries are provided (such as SIP, Diameter – Cx/Dx, Sh, and DCCA). The applib, as each one of these libraries is usually referred to as, provides the base TTCN-3 component definitions and functions. Any load test application can be developed by choosing the appropriate applib.

Control Logic The control logic realizes the executable part of TITANSim. It is built on top of the CLL and the applib. It is application specific and ensures that the expected traffic flow is realized - as shown in Figure 13.

Run time GUI The run time GUI offers a user interface with which a tester can control the test's execution. Using the GUI, the tester can control logging options, view test statistics, and start and stop test execution.

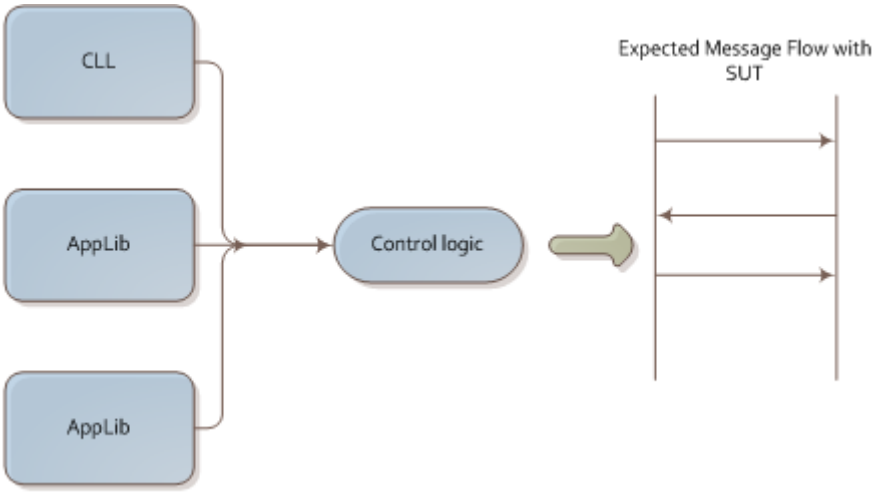


Figure 13: Control logic

4.2 TITANSim Component Structure

Each TTCN-3 component is similar to an object in any object oriented programming language. Each component has a set of functions and component variables that are declared in the component definition. To use a feature provided by a particular component, one extends that component, and use the functions and variables provided by it. Any system test load application developed on TITANSim will have a component structure similar to Figure 14.

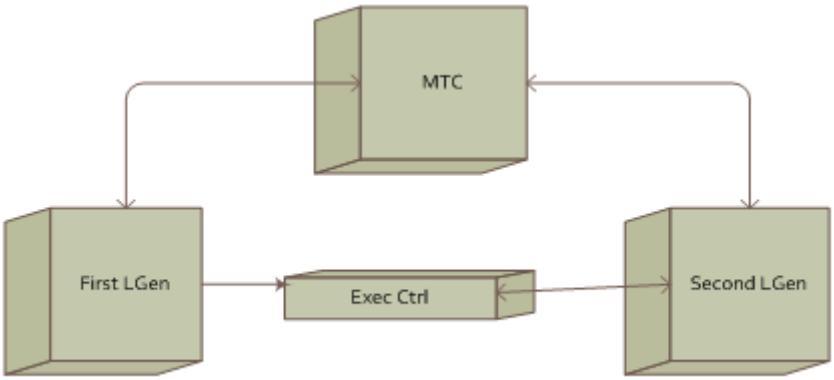


Figure 14: TITANSim component structure

The MTC is created by the TITAN framework offered by the TITAN libraries. The test cases designed for a particular load application start executing inside the MTC. The actual load generation (i.e., the traffic itself) is created by the Load Generator components (LGens) which execute within a PTC. Depending on the test design many LGen components will be created during the test execution, hence an Execution Control (ExecCtrl) component is used to initiate and oversee the traffic execution and synchronize the finish events of these LGens.

The TITANSim CLL provides a LGen base component, which can be extended by a specific load generation application. The LGen base feature forms the heart of the TITANSim. It enables the concurrent execution of many PTCs. Each PTC is mapped to an OS process. So in order to generate high traffic load conditions, theoretically we need many PTCs which make the load application quite resource intensive and inefficient. However, the LGen base feature is designed to simulate multiple users with a single PTC. By simulating hundreds and thousands of users on a single PTC, the number of OS processes required to generate the traffic load is significantly reduced. The design of such a LGen base feature requires a new programming paradigm called *event driven programming* which is discussed in the following section.

4.3 LGen base Programming Paradigms

Traditionally two programming paradigms are frequently used in a TTCN-3 test system design. They are, sequential programming and event driven programming. Sequential programming is used in test system design for *functional* testing, where usually only one user

is simulated by the test system in each test case – because the emphasis in functional testing is on correct execution of each function that is being tested. In contrast event driven programming is frequently used in test system designs for a system test, as there is generally a need to simulate a million users in each test case.

4.3.1 Sequential programming

In sequential programming, a set of programming instructions are defined in the test system and order of the execution is fixed – as shown in Figure 15.

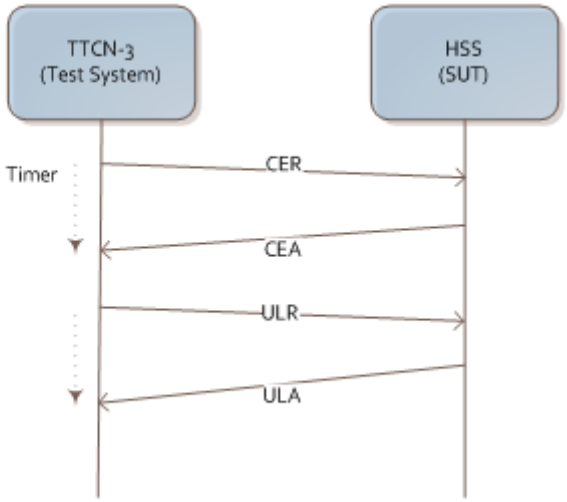


Figure 15: Sequential, hardwired TTCN-3 test system

Figure 15 showed a sequence of Diameter messages exchanged between the TTCN-3 test system and the HSS. In this case the execution is strictly sequential and the test system waits for some maximum period of time for a response from the HSS. The main drawback of sequential programming is that the execution of the test system is blocked until it receives a response from the SUT or a timer expires, *for each message* sent out by the test system. So with a naive mapping of the test generation of each individual user to a single process, this technique can simulate only one user per PTC. This technique is not scalable; hence it is not suited for system testing high capacity nodes such as the HSS, which must handle thousands of CPS.

4.3.2 Event driven programming

Instead of a fixed sequential execution behavior event driven programming enables dynamic execution order; hence the test system behavior is described in terms of actions assigned to events. Whenever a particular event occurs, the test system executes the corresponding assigned action. Therefore the test system does not wait for a response from the SUT, but rather it reacts to the events from SUT by executing the defined actions for a particular event, as shown in Figure 16. This facilitates the test system interleaving the execution of a very large number of users while only needing to maintain the necessary state for each of these users – but *without* the overhead of a process per user.

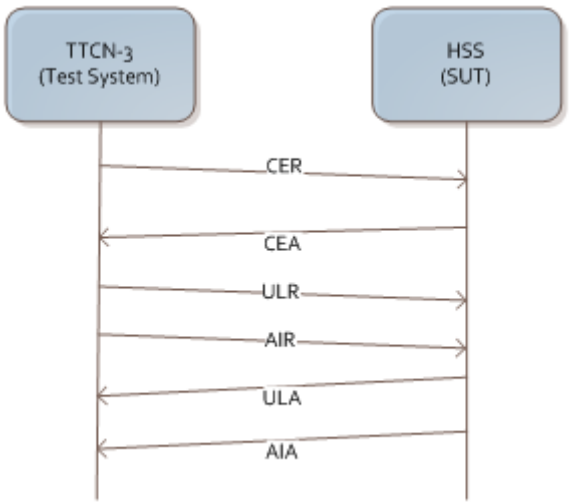


Figure 16: Event driven, dynamic TTCN-3 test system

Thus in event driven design a PTC does not block the execution of the test system, so instead of just a single user per PTC – multiple users can be simulated using each PTC. So a single PTC sends request messages for a number of users to the SUT and handles the response messages for each of these users as and when the response arrives from the SUT. The ideal design approach is to have as many PTCs as the number of CPU cores, say 2 PTC's for duo-core and 4 for the quad-core processor system.

4.4 Finite State Machines

The dynamic behavior of TTCN-3 test systems designed using the event driven programming paradigm, can be better explained with the use of the concept of FSMs. A FSM is described with a set of events, test steps, and states. Every event will have an assigned test step. At any point of time during test execution the test system is in a particular state of the FSM and listens for the events that are expected to occur in that state. Events could be timeout messages, responses messages from the SUT, and so on. Upon the occurrence of a particular event the test system executes the corresponding test step and moves to the next state (or could even remain in the same state). For example, consider a FSM consisting of two states (S0 and S1), two events (E1 and E2), and test steps (TS1, TS2). When the test system is in state S0, if the event E1 occurs it executes TS1 and moves to S1. Similarly if the test system is in S1 and if event E2 occurs, then it executes TS1 and moves to state S0. FSMs can be described in the form of a table, as shown in Table 2, or in a state diagram (i.e., in a graphical format) as shown in Figure 17.

Table 2: A sample FSM table

Event/ next states	S0	S1
E1	TS1, S1	TS2, S1
E2	TS2, S0	TS1, S0

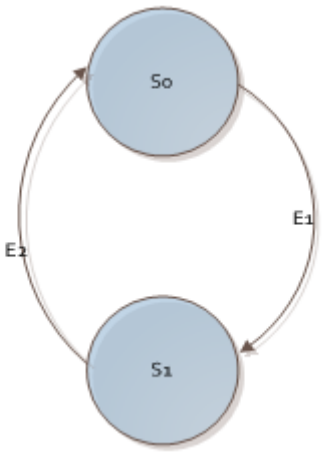


Figure 17: State diagram

4.5 Scheduling concepts

Since the TTCN-3 test system involves many PTCs which act as LGens there is a need for a base feature that takes care of LGen component initialization, test case execution, handling the result of each traffic case execution, enable/disable a LGen component, and finally clean up all the LGens and the associated data bases when the test case execution is complete. The TITANSim CLL has a base scheduling feature that provides all these functions. It can be used to schedule the PTCs used in the test system design based on the two different programming paradigms discussed in previous section. The base scheduling feature provides two types of scheduling methods: Central Scheduling (CS) and Distributed Scheduling (DS) [6].

The CS method is based on the sequential programming paradigm, where the test system is likely to consist of hundreds of PTCs as shown in Figure 18. In the central scheduling architecture based load test application, the scheduling of test components is centralized which results in a lot of communication overhead within the test system, as hundreds of test components communicate with a *master scheduling* component during the test execution.

The DS method is based on the event driven programming paradigm. Hence, the test system utilizes only few PTCs. In this case the scheduling functions are distributed and performed by the LGens themselves in each PTC, rather than in a master scheduling component. The DS method is shown in Figure 19.

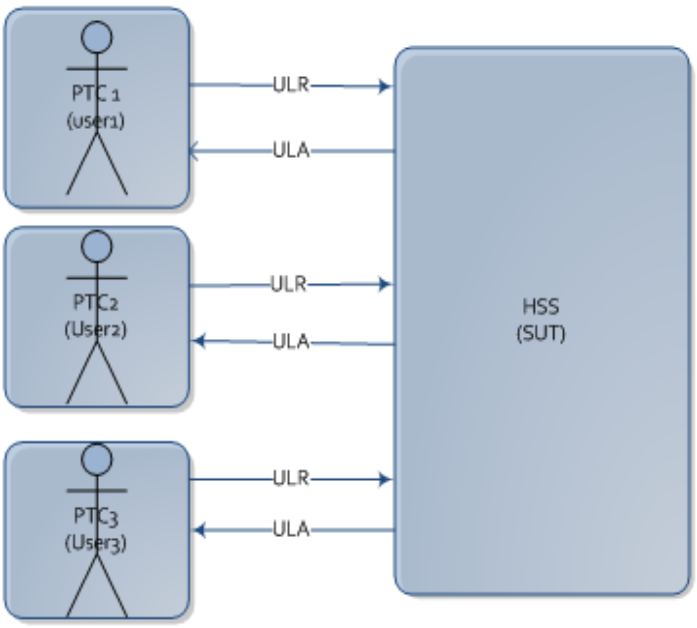


Figure 18: Central Scheduling

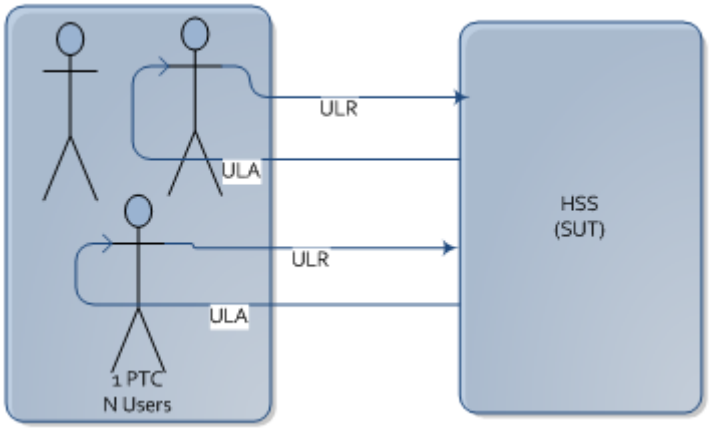


Figure 19: Distributed Scheduling

Table 3 summarizes the differences between the CS and DS scheduling.

Table 3: CS versus DS

Central Scheduling	Distributed Scheduling
Sequential test system	Dynamic, event driven test system using FSM
One user simulated in each PTC	Thousands of users simulated in each PTC
Typically uses hundreds of PTCs	Ideally the number of PTCs equal the number of processor cores
Uses a master scheduling component	The scheduling functions are distributed and handled

	by each PTC
Internal communication overhead	No overhead
Traffic cases implemented in TTCN-3 modules	Traffic cases implemented in FSMs (configuration files)
Not suitable for system test. Typically used for functional testing purposes	Highly suited for system test, consumes less test system resources, yet capable of generating high traffic load conditions

4.6 Distributed Scheduling architecture

The concept of Distributed Scheduling based on FSMs was proposed by Ferenc Bozóki and Tibor Csöndes of Ericsson Hungary in their publication titled '*Scheduling in Performance Test Environment'* [27]. As discussed in section 3.4, Ferenc Bozóki and Tibor Csöndes extended the concept of a thread to an abstract level called *virtual threads*, where a single thread could emulate multiple users and the execution of each user is considered as a virtual thread. Figure 20 (adapted from Figure 3 in page 2 of [27]) shows one such thread executing User a, User b, and User c as separate virtual threads. The execution behavior of each user is dynamic and it is defined by a FSM as described in section 4.4. Each thread holds a database to store the relevant information about the users. Each record of this database corresponds to a specific user. Thus, by simply *varying the index* of this database, the execution context changes to a different user. Compare this to the case of central scheduling, where the *OS scheduler has to make context switches* between processes to change the execution context to another user.

Each thread holds an event queue which is a simple FIFO queue [30] for storing the incoming events. The message in the queue is processed to determine the user it belongs to, and the message is dispatched to the appropriate user using an event dispatcher. The user then performs the action according to the assigned FSM. Each thread also maintains a timer queue which holds the timer events of the users. The thread adds a timer event to the queue for a specific user only if it is required; say for instance if a response to an event has to be sent within two seconds. When such a timer expires in the timer queue, the timeout event is dispatched to the corresponding user. A bottleneck with this approach is that certain events in the queue might require computationally intensive operations, i.e., use significant CPU time. This would deprive the other events in the queue (which may have a comparatively smaller timeout value in the timer queue) of CPU time and may lead to unnecessary timeout events.

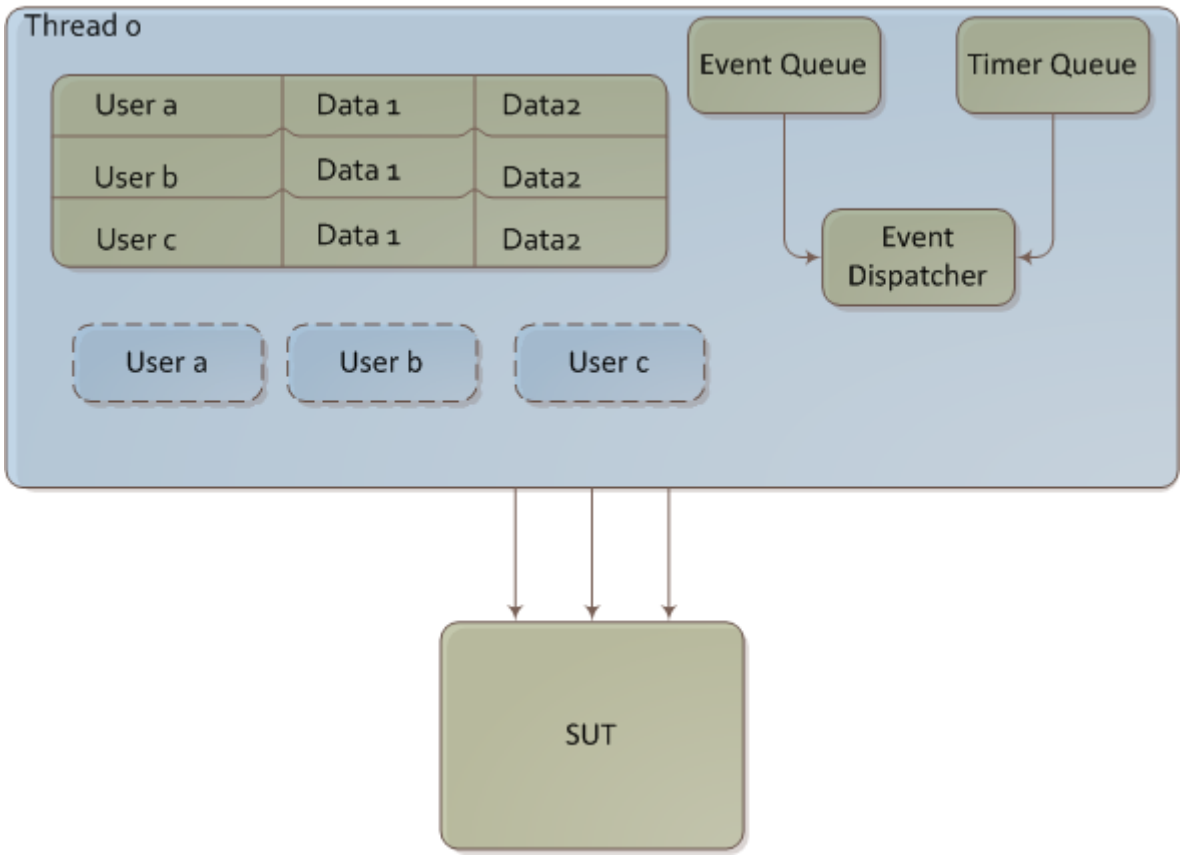


Figure 20: DS architecture[††]

As almost every message exchanged between the test system and the SUT is timer driven, the thread has to process a message within the appropriate time; otherwise the message is considered to be too late (leading to timeouts) leading to a false (test) verdict. Thus, there could be undesirable delays in processing of events in the test system as only a single thread is responsible for handling events for typically thousands of users. Therefore, Ferenc and Tibor proposed an algorithm to reduce the total delay of the test system by optimizing the scheduling of events by introducing a secondary queue. Using this algorithm the thread *prioritizes* the event queue and determines those events that can be put on hold temporarily (in the secondary queue) without causing an *actual delay* (as those events have a higher timeout value compared to the rest in the event queue) in the test system. For detailed discussion of this algorithm refer to ([27], page 4).

4.7 The LGen base – finer aspects

In TITANSim terminology any user that is simulated by the PTC is referred to as an entity. Henceforth we will use the term entity to refer to any object simulated by the TTCN-3 test system. A simulated entity can be configured to exhibit different behavior types. For example, all the SIP related capabilities are referred to as the SIP behavior type, similarly Diameter protocol related capabilities are referred to as the Diameter behavior type. An entity assigned a list of behavior types is called an entity type. A group of entities of same type

[††] Adapted from Figure 3 in page 2 of [27]

form an entity group. Thus an entity group consists of several instances of entities (such as users) exhibiting common behavior types. Since the system behavior is described by FSMs, the behavior type assigned to entities consists of a set of test steps and events. Once the behavior type with test steps and events is assigned to the entities, a traffic case is created. A traffic case basically defines a certain flow of traffic (as described in the FSM) assigned to the entity. Depending on the requirements of the load application, a number of traffic cases can be created, by assigning different behavior types to different entity types. Once a traffic case is created, it is important to define how much traffic of this kind should be generated by the test system. Hence, the required traffic rate or load level, and the number of PTCs to be used are configured by creating a scenario. Figure 21 shows the relation between entity, behavior type, and the FSM.

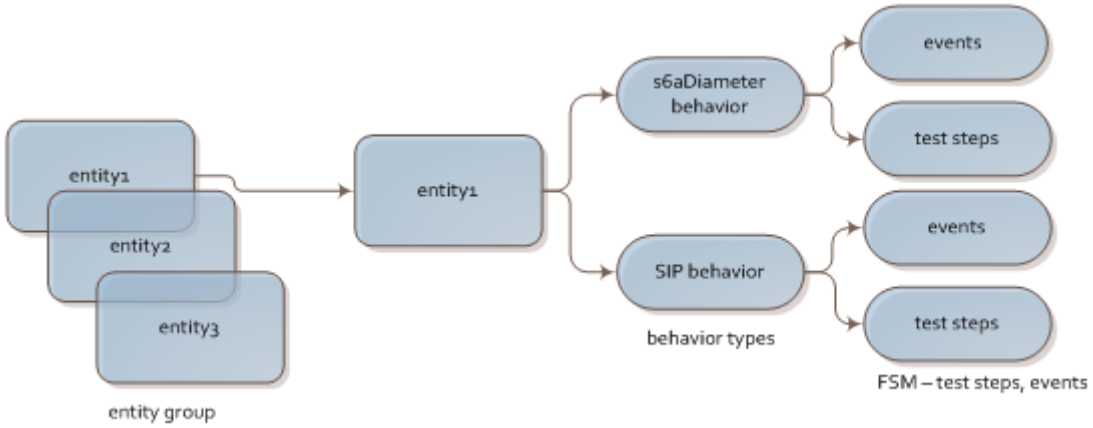


Figure 21: Relation between entity, behavior type, and FSM

5 Methods

This chapter presents the environment for developing the load test application, the S6a load application design and implementation, and the test system setup used for the tests executed with the S6a load application and their results.

5.1 Goals

This project aims to create a Diameter load application based on the distributed scheduling architecture of TITANSim, specifically for the S6a interface. The scope of the load application is limited to the Initial Attach traffic scenario which will be described in later sections. A thorough investigation is made of how to deploy the solution in a real system environment[‡‡] (including using separate traffic generators, with a real target node running the HSS software as the SUT). Performance measurements and a comparison of the earlier CS based solution with the new DS based solution.

5.2 Setup

Figure 22 shows the environment used to develop the load application. It consists of TITANSim, a Diameter Proxy, and the HSS application installed and running in a simulated environment called Maia[§§].

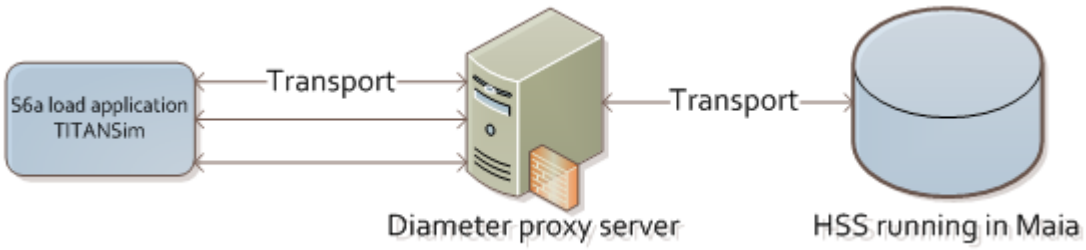


Figure 22: The development setup

This figure shows the logical view of the setup for the sake of clarity. Physically all of these elements are run as separate processes in a single Linux workstation running SUSE[***] Linux 10.1 Desktop 32 bit edition.

5.2.1 TITANSim setup

The TITANSim software is developed and maintained by TCC (Test Competence Center), a Research & Development organization focusing on TTCN-3 within Ericsson's unit in Hungary. All the official releases of the software are available for download only internally within Ericsson. For this project the latest release of the TITANSim software called the

[‡‡] The load application was designed and tested only in a simulated environment during its development cycle. Hence additional investigation is needed to deploy the load application in real environment.

[§§] Maia is an Ericsson proprietary tool used to simulate the Ericsson Telecom Server Platform (TSP), in which the HSS application is installed and run. For more information, see http://www.ericsson.com/ericsson/corpinfo/publications/review/2001_04/files/2001045.pdf.

[***] An open source Linux distribution, see http://www.opensuse.org/en/

EPTF GenApp (Ericsson Performance Test Framework Generic Application) version R7A03 is used. This version of the software provides all the required libraries needed for developing load applications for Diameter traffic type. As discussed in chapter 4, TITANSim is written exclusively for execution with TITAN, the proprietary TTCN-3 execution platform used within Ericsson. The latest version of TITAN software, version 1.8.pl4 was installed on the computer as it is a prerequisite for TITANSim. TITAN is also available as a plug-in to the Eclipse SDK, which is used for TITANSim development. A screen shot of the TITANSim development and execution environment as an Eclipse plug-in is shown in Figure 23.

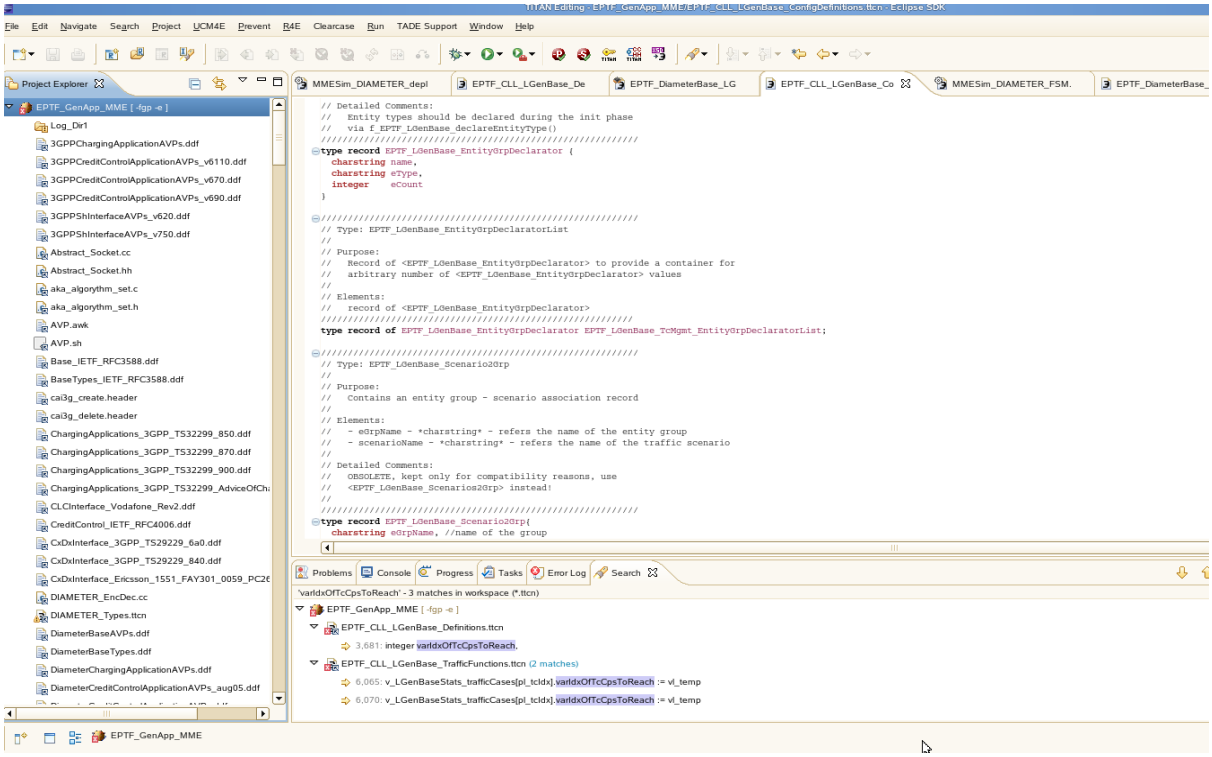


Figure 23: TITANSim EPTF GenApp in Eclipse SDK

5.2.2 Diameter Proxy

In actual operation, the HSS does not handle more than two or three transport connections per peer node. However, in a typical system test scenario where there may be multiple traffic generators used in the test system, thus there is a need for the HSS to support at least one connection per traffic generator. Unfortunately, the limited number of connections supported by the HSS is not sufficient to support the required number of traffic generators needed for the system test. To circumvent this problem a Diameter Proxy is used. The Diameter Proxy is a tool that acts as a proxy server between the traffic generators[†††] and the Diameter server under test (in this case the Diameter server is the HSS). The Diameter Proxy of the HSS establishes a transport connection with the HSS and supports a unique transport connection

[†††] These traffic generators are Linux workstations that run the TITANSim load application during a typical system Test, usually one or more traffic generators are used in the system test depending on the required load levels.

with each of the traffic generators (Diameter clients). All of the traffic generators address their messages to the Diameter Proxy, which performs multiplexing/de-multiplexing of the messages between the HSS and traffic generators, as shown in Figure 24.

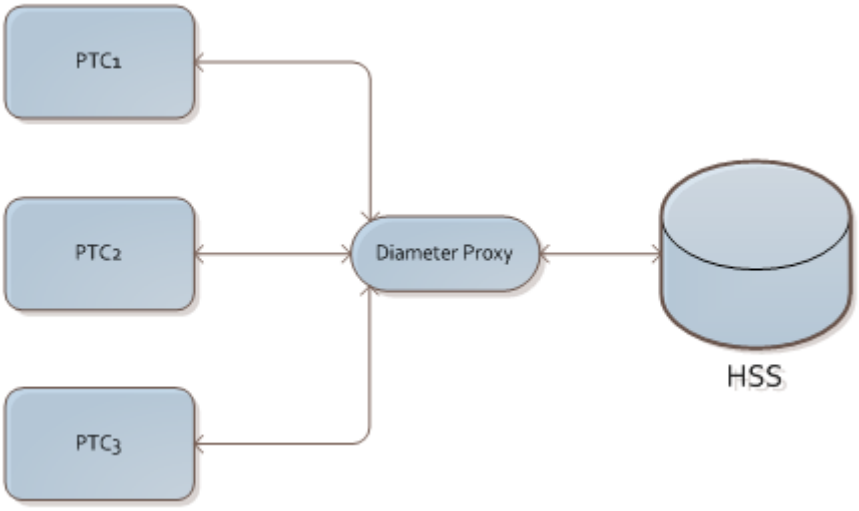


Figure 24: The Diameter Proxy architecture

The Diameter Proxy exchanges CER-CEA messages with the HSS and also supports DWR-DWA messages to keep the connection alive. It uses a separate thread to serve each of the clients (the TTCN components) connecting to it and uses the user name AVP to uniquely identify and associate a connection with the corresponding client. Table 4 shows a typical configuration of the Diameter Proxy server setup for this project. Figure 25 shows a sample screen shot once the Diameter Proxy is up and running.

Table 4: Diameter Proxy Configuration

Main configuration parameters	Value
Supported node	hss
Diameter server IP address	192.168.83.100
Diameter server port	3870
Protocol	tcp
No. of Diameter server connections	1
Diameter Proxy port	3876

```
enirkal en maia03 [bin][48]> ./DiaProxy4.0 -lm 63 -p 3870

Configuration read successfully

logging to /home/enirkal/DiaProxy_maia03.log

DiaProxy version 4.0

Your current configuration file:        DiaProxy.cfg

CER settings
------------
Supported node                  : hss
Origin Host                     : Origin0.ericsson.se
Origin realm                    : ericsson.se
Product name                    : Ericsson Diameter
Vendor ID                       : 0
Supported Vendor ID             :
Auth application ID             :
Vendor Specific application ID  : 10415-16777216,10415-16777251

Node settings
-------------
Diameter Server IP address      : 192.168.83.100
Diameter Server port            : 3870
Diameter Server Protocol        : tcp
Number Diameter Server Con.     : 1
DiaProxy port                   : 3876
Log mask                        : 0x3F (63)
Monitor time                    : 0
Inactivity time                 : 60
Max number of connection retries: 1

************************************************
************************************************
(ProxyThread) :   CONNECTING......
************************************************
************************************************
(DiaThread:3065158560) : Trying to establish connection to Diameter:192.168.83.100:3870 using TCP

(DiaThread:3065158560) : Network connection to 192.168.83.100:3870 established

(DiaThread:3065158560) : CER-CEA connection established with Diameter.

(ListenerThread:3063057312) : Listener Thread starting up

(ListenerThread:3063057312) : Server side is listening on port 3876

************************************************
************************************************
(ProxyThread) :   DIAPROXY UP & RUNNING
************************************************
************************************************
```

Figure 25: Diameter Proxy

5.2.3 HSS setup with Maia

Ericsson employs carrier-class technology when it comes to developing telephony and server applications. Ericsson's Telecom Server Platform (TSP) is a carrier class server technology for multimedia applications and control functionality. TSP provides very high availability and enables operators to deploy different functions on the same hardware and to save cost [20]. For example, functionally P-CSCF, S-CSCF, and I-CSCF are different nodes, but all of them can be installed on a single TSP and run in real time as functionally different nodes, even

though they are all installed in a single physical computer. A variety of nodes developed by Ericsson such as HSS/MGCF/CSCF in the IMS and SCF/HLR in the GSM, GSN in GPRS run on TSP.

TSP is built on component based architecture and uses commercially available components to build its structure. The architecture of TSP is shown in Figure 26. It consists of a cluster of processors running Linux and the DICOS‡‡‡ operating systems forming the operating system and hardware layers. The heart of TSP is the middleware layer provided by the TelORB clusterware. It connects the different processors and facilitates using Inter Processor Communication (IPC). The TelORB clusterware also has an object-oriented database providing persistent storage in RAM. The database is distributed over the processor cluster. Thus each of the database items is replicated and always stored in more than one processor. As a result of this replication an application running on TSP is highly robust and protected against data loss even if a processor crashes at run time. The node management layer supports standards such as CORBA, LDAP, HTTP, and SNMP; and enables the easy integration of the TSP node with external management systems used by the operator. The signaling layer supports SS7 and Diameter stacks over IP. It is a highly scalable layer and can be distributed over multiple processors called Traffic Processors (TP). While the TPs handle the real time traffic, Input/Output (I/O) processors handle the operations and maintenance operations performed in the node. TSP is available in a number of configurations depending on the size of processor cluster (Maxi-42, Midi-31, Mini-21, and Micro-10). The processor cluster is also logically divided into many different processor pools for the sake of distribution of software into different processor pools [18].

During the system test the HSS application is installed and configured on a real TSP node referred to as the target node (in this case it is the SUT). Maintaining a complex node such as the TSP is a very expensive and complex process, hence the target node is only available for a certain window of time for each system tester within Ericsson. As a result this target node is only used for performing actual system tests. So, all development activities based on TSP utilize Maia (a TSP simulator). Maia simulates both an I/O processor and one or more TPs on a single Linux workstation using VMware§§§ virtual machine software. Therefore, a VMware license is a prerequisite for running Maia on a Linux workstation. Each simulated I/O processor and TP corresponds to a separate VMware virtual machine instance. Maia supports most of the functions actually supported by the TSP and provides the user with very TSP-like environment. All the functions of TSP simulated by Maia are invoked using the **epmaia** command. Figure 27 shows the execution of the different epmaia commands involved in setting up Maia.

In order to setup the Maia configuration, **epmaia setup** is executed. This creates the necessary VMware configuration files for the I/O processor(s) and TPs, SS7 configuration files, node management (NM) configuration files, site database, and other TSP configuration files. In TSP each executable software module is packaged in a structure called a Load Module (LM). The HSS application that is loaded on TSP is also organized into a number of

‡‡‡ DICOS is an Object Oriented RTOS developed by Ericsson
§§§ A virtualization software, check http://www.vmware.com/virtualization/

LMs. Thus the base TSP software (Maia LMs) is organized into a number of LMs that take care of separate functions such as IPC, SS7, and Diameter stack functions.

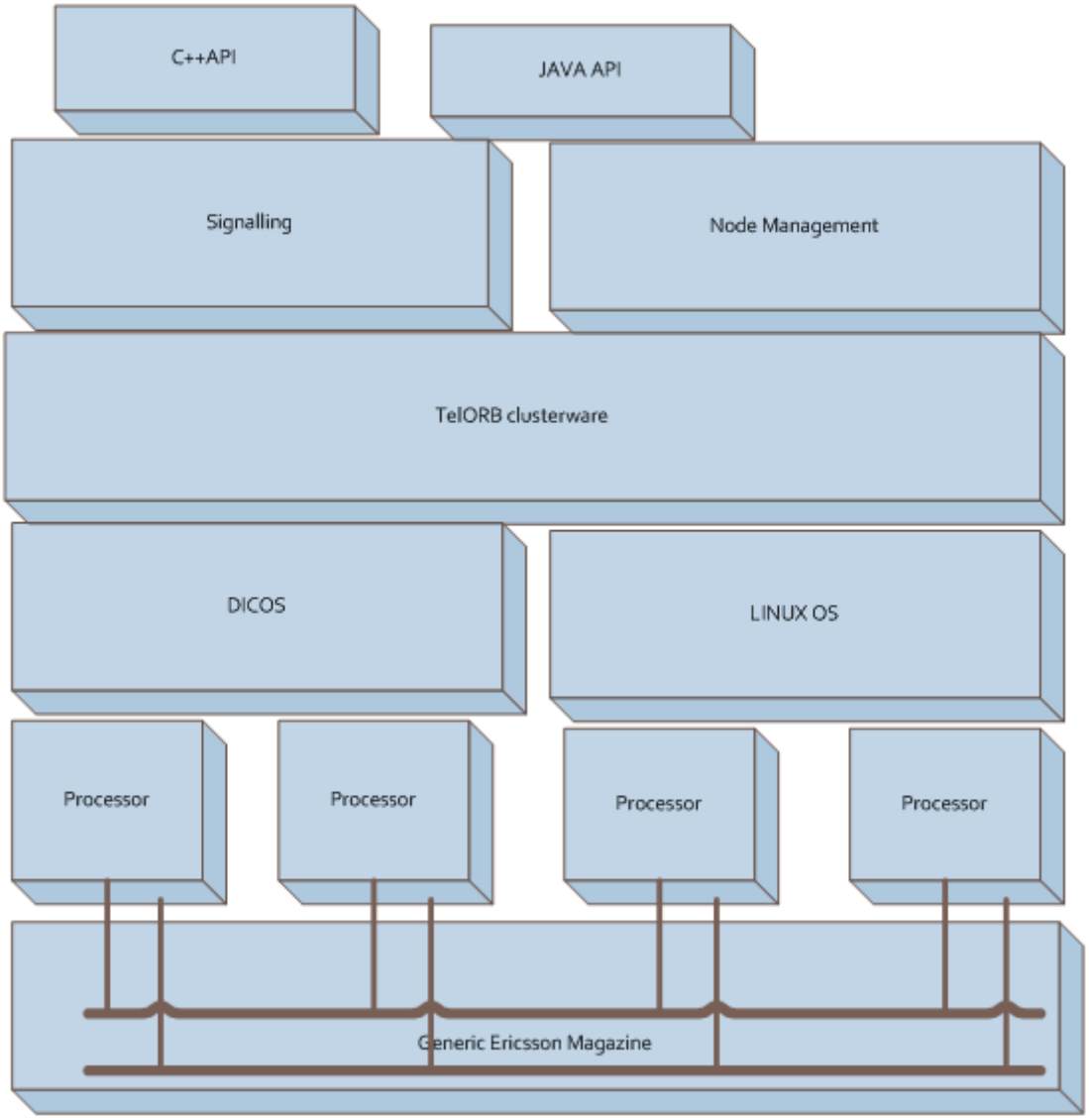


Figure 26: The TSP architecture

The HSS application LMs perform functions such as HSS subscriber data access, HSS subscription management, etc. For this project LMs from official TSP release 6500 and HSS release LSV3 are used. Each of these LMs also contains a configuration file called the **epct** file that specifies in which processor pool this LM should be primarily loaded and in which processor pool it should be replicated. The loading of appropriate LMs into the corresponding processor pool is performed by the **epmaia epct** command which results in the generation of the Loading Group. The I/O processor is then started by executing the **epmaia start** command. The generated configuration files for NM, SS7, and site database are then loaded from the host machine into the virtual I/O processor is done by the **epmaia upload** command. The TPs are then started using the **epmaia start** command. The simulation environment can be stopped using **epmaia stop** command by specifying the individual processor name or

using the option **-all** to stop all processors. The status of the Maia configuration and the VMware virtual machines can be checked using **epmaia status** command. The command **epmaia save** can be used to save log files from the console, daemon logs, and crash dumps from the I/O. Each of the epmaia commands has a number of options that can be used depending upon on the needs of the tester. TSP uses the virtual IP (VIP) concept; hence the entire processor cluster can be addressed by one VIP address. Table 5 shows the IP address used in Maia. A sample screen shot in Figure 28 shows the processor TP2 start up following the Maia setup.

Table 5: Maia IP addresses

Symbolic name	IP address
IPv4 VIP external	192.168.83.100
IPv4 VIP internal	192.168.53.100
I01	172.16.8.100
TP1	172.16.8.102

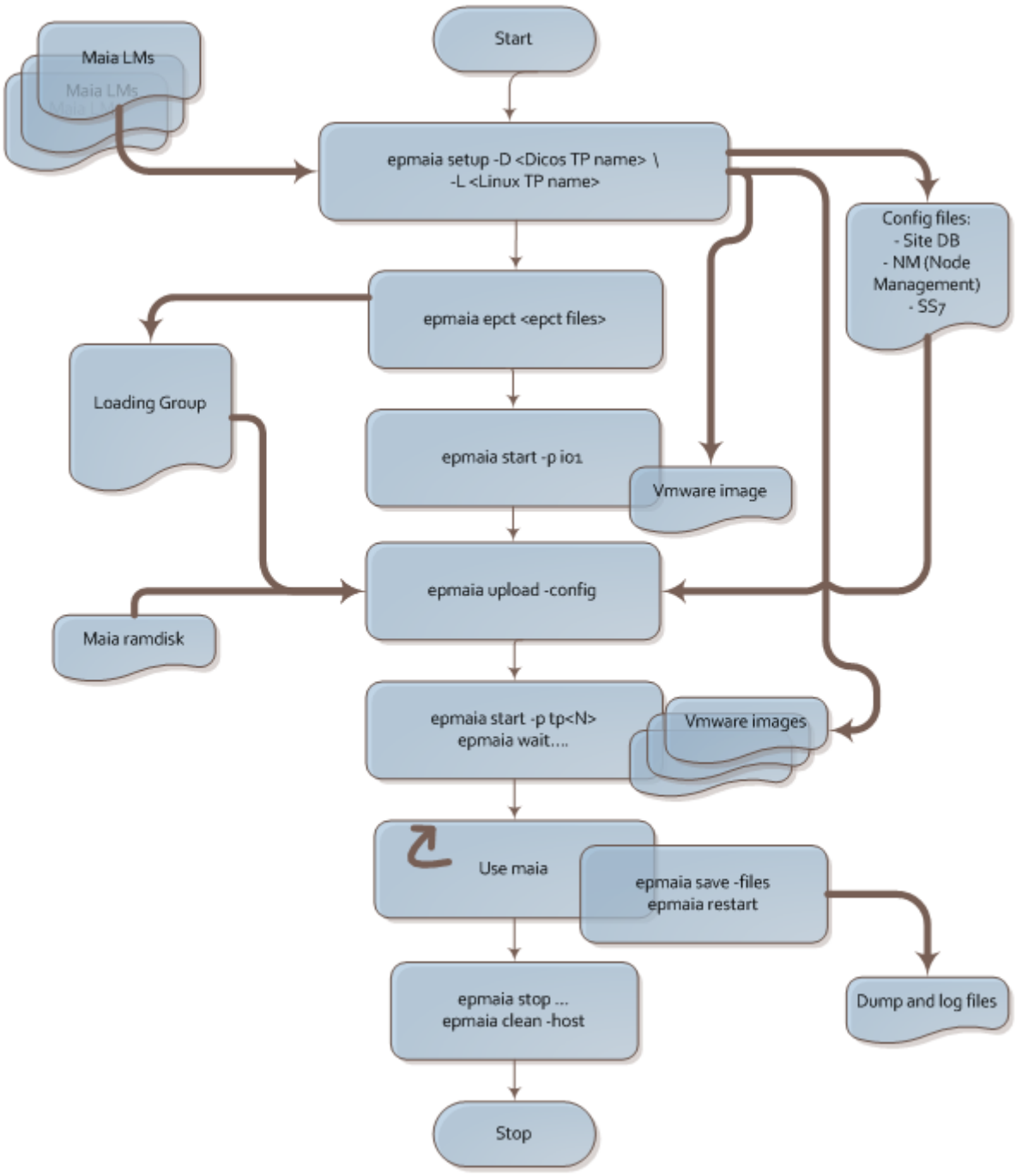


Figure 27: Maia setup[****]

[****] As simple as it might sound on paper, the Maia setup process is actually quite complex. A number of issues were encountered and solved regarding the Maia setup during this project. These issues were mainly related to determining the correct epct files to use for the corresponding release of TSP and HSS. Other issues were related to provisioning of the Maia (HSS) database with subscriber information. Some key database objects necessary for provisioning HSS were not installed in Maia properly. Resolving all of these issues took roughly 3 weeks. After this Maia was up and running and the HSS was successfully provisioned.

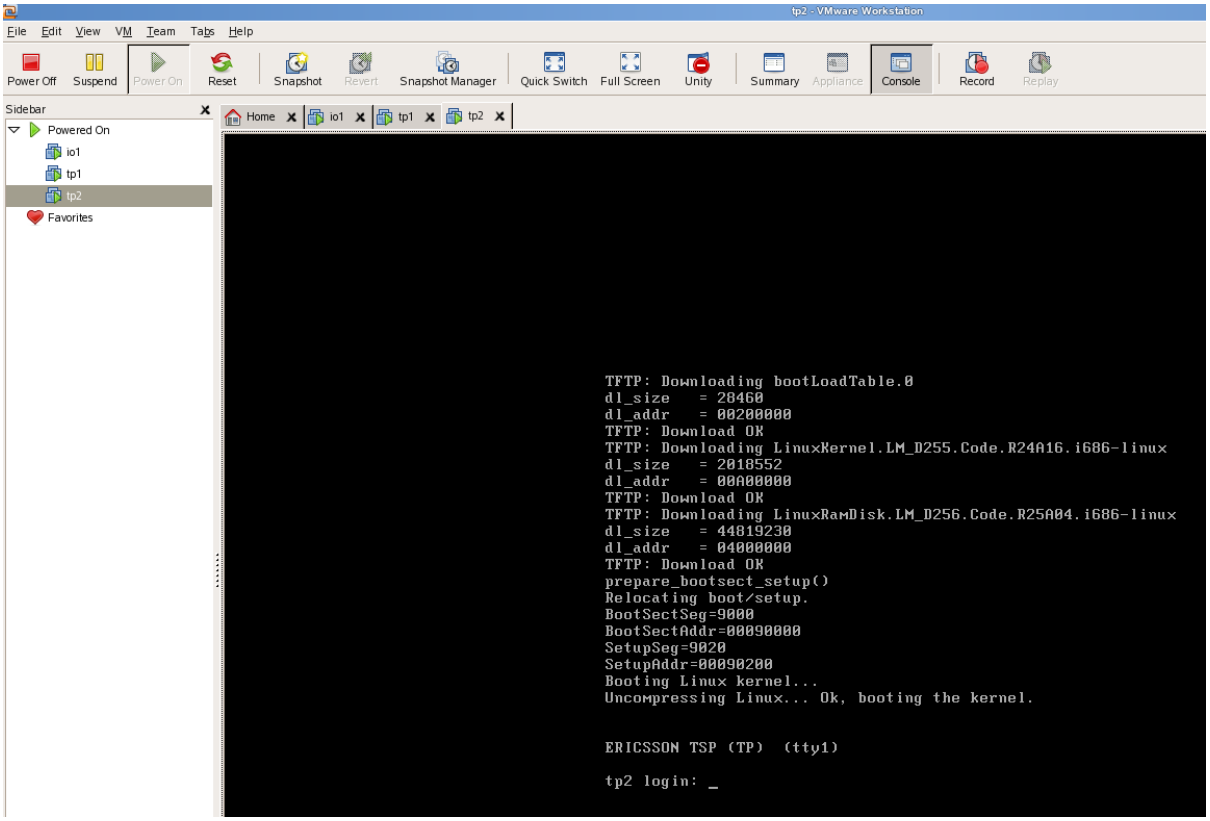


Figure 28: TP2 startup after Maia setup

5.2.3.1 HSS Provisioning

Once the HSS is successfully running after the Maia setup, the next step is to configure the HSS and provision it with all the necessary subscriber information. Since we have chosen to use the ESM module for this project, all the configuration and provisioning is performed solely to run the ESM application in HSS (Maia). This provisioning is performed using an internal tool called “population centre”. The population centre tool uses a number of shell scripts to set the configuration parameters in order to set the ESM Diameter stack up and start it running in the HSS (Maia). Once the ESM stack is configured the database container objects that hold the subscription information are populated. All the configuration and database population are performed through the Lightweight Directory Access Protocol (LDAP) [19] interface offered by the TSP. Figure 29 shows the LDAP browser tool which is used for manually setting up the LDAP parameters in any TSP node. Once the provisioning process is complete, the HSS will be ready for traffic for the system test.

Maia introduces certain limitations to HSS provisioning. In the case of a typical target node in a system test roughly 100 000 ESM subscribers are populated in the node. However, since Maia is being used as our simulator it imposes certain memory limitations. Therefore to keep the HSS application stable and prevent the Maia process from crashing, the provisioning was limited to roughly 10 000 ESM subscribers.

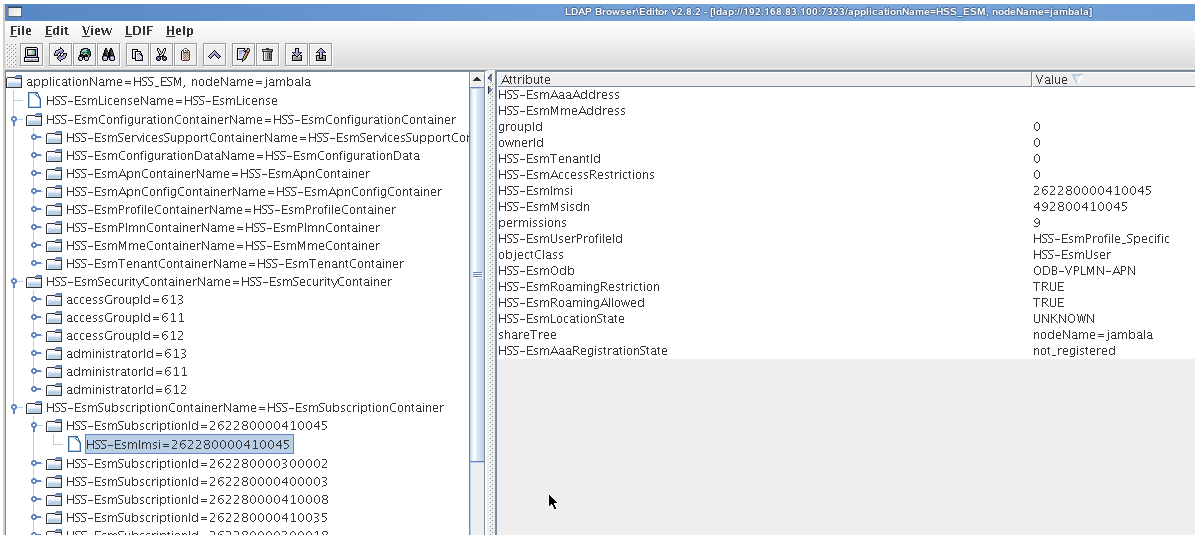


Figure 29: LDAP browser showing the ESM stack and ESM subscriber parameters

The population centre tool also takes an input configuration file through which we can control the number of ESM subscribers that have to be populated along with certain other node parameters. Figure 30 shows the population centre tool that is used for HSS provisioning.

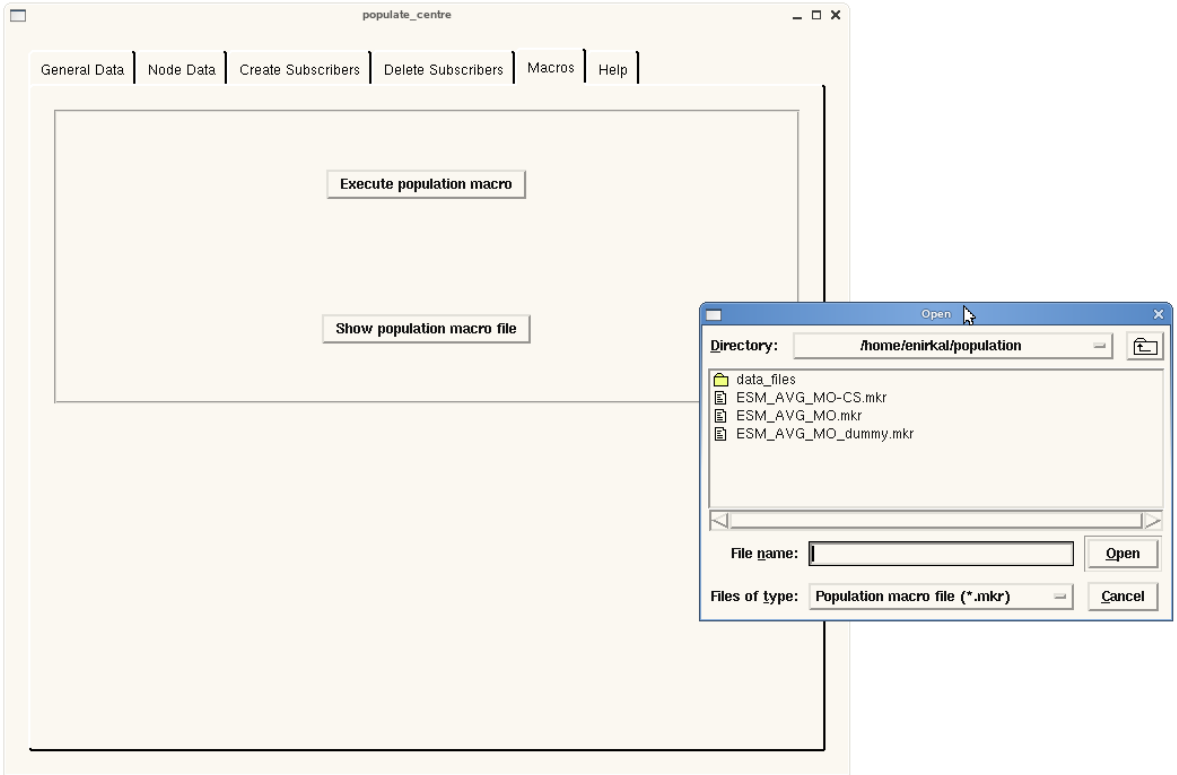


Figure 30: Population centre tool

5.3 Architecture of the proposed load application

Figure 31 shows the architecture of the proposed S6a load application. The proposed solution consists of a new S6a LGen component which basically extends the DiameterBase LGen component and DiameterBase Transport component provided by the TITANSim applib. The DiameterBase LGen component in turn is an extension of the LGen base component feature provided by the TITANSim CLL as discussed in section 4.2. The other functions such as Statistics, Logging, Execution control, and RT GUI for the application are provided by the standard TITANSim CLL. A detailed description of the S6a load application design is provided in the following section.

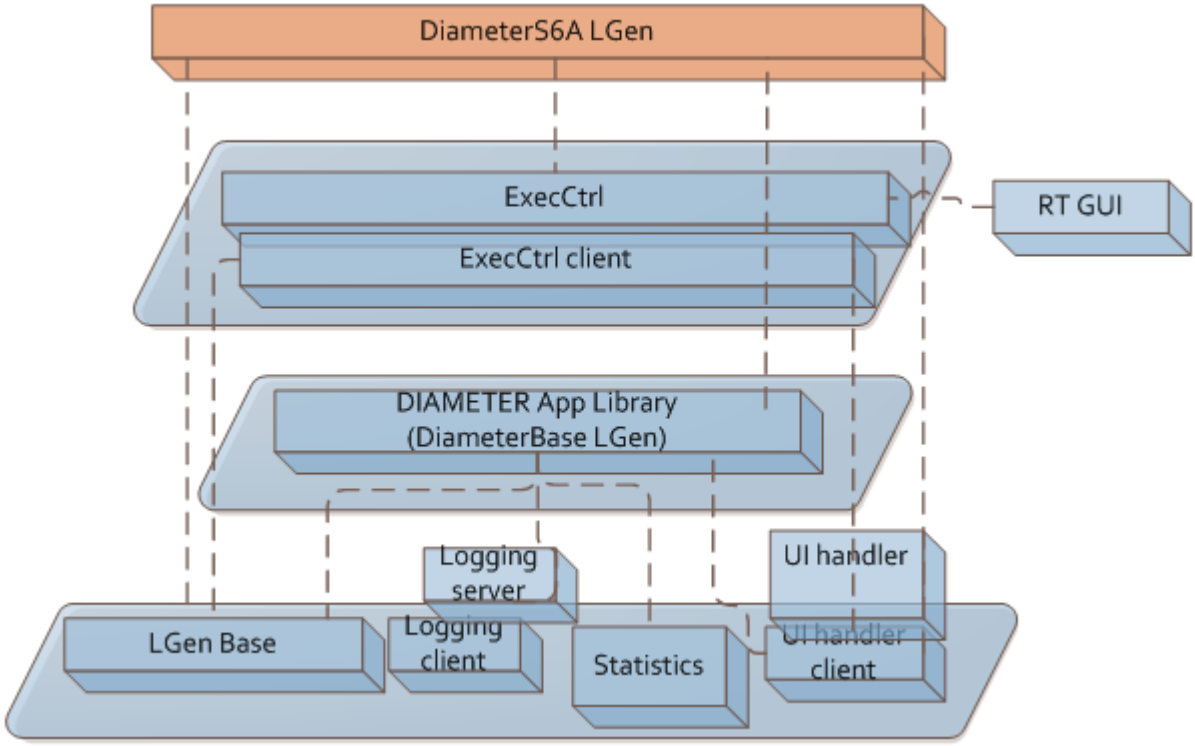


Figure 31: S6a application architecture

5.4 Solution design

5.4.1 The traffic case

After the provisioning of the HSS the Diameter Proxy is started. The Diameter Proxy establishes a Diameter session with the HSS by exchanging the CER/CEA messages. Similarly, when the S6a application (simulates the MME) is started it establishes a Diameter session with the Diameter Proxy – as shown in Figure 32. After the Diameter session is established between the nodes the Diameter traffic (such as the S6a) can be run over the Diameter session.

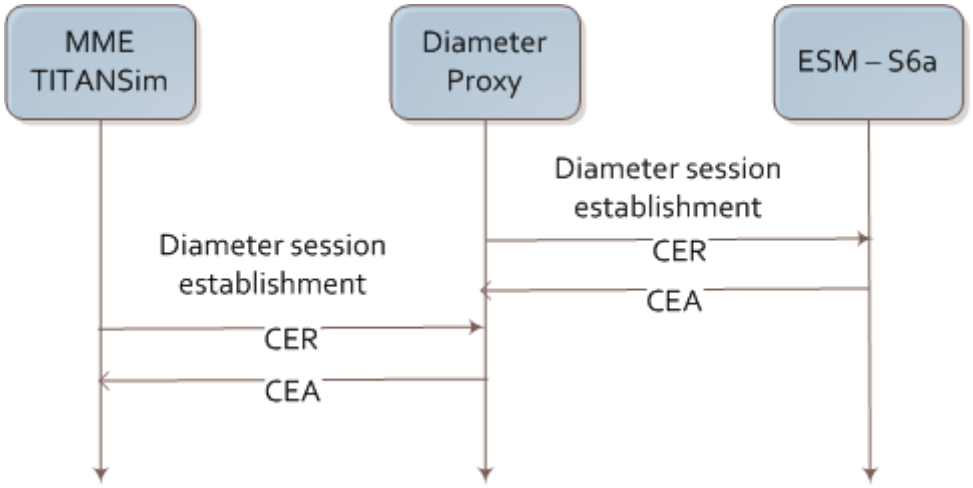


Figure 32: Diameter session establishment

As briefly discussed in chapter 5.2.3, the focus is mainly on the ESM module within HSS defined by the S6a interface [13]. To be specific, the scope of the load application for this project is limited to the Initial Attach traffic case, although the application could be extended in the future to support other traffic cases. Figure 33 shows the traffic flow diagram of the Initial Attach traffic case followed by a brief description. For a detailed description of the messages with the Diameter Attribute Value Pairs (AVPs) involved, refer to [13].

The Initial Attach procedure is carried out between the MME and the HSS over S6a interface in order to authenticate a UE located in the LTE network. The MME requests authentication information from the HSS using AIR/AIA messages. Upon receiving the AIR message, the HSS checks if the IMSI of the user is known and requests the Authentication Centre (AuC) to generate Authentication Vectors (AVs). The HSS then returns the result code AVP DIAMETER_SUCCESS (2001) in the AIA message along with the generated AVs to the MME.

If the MME wants to update any of the user's identity stored in the HSS, it can do so by sending an ULR message to the HSS by including the IMSI and the supported Radio Access Technology (RAT) types. Upon the reception of a ULR the HSS checks if the supported Radio Access Technology (RAT) type in the request is allowed for this subscriber. It then checks if roaming restrictions are applicable to the user in this specific Visited Public Land Mobile Network (VPLMN). Then based on the operator's configuration to allow or bar the user from accessing the packet oriented services according to the user's current location (i.e., based upon the Home Public Land Mobile Network (HPLMN) or the VPLMN) the HSS chooses to include or not include the subscriber's Access Point Name (APN) configurations in the ULA message.

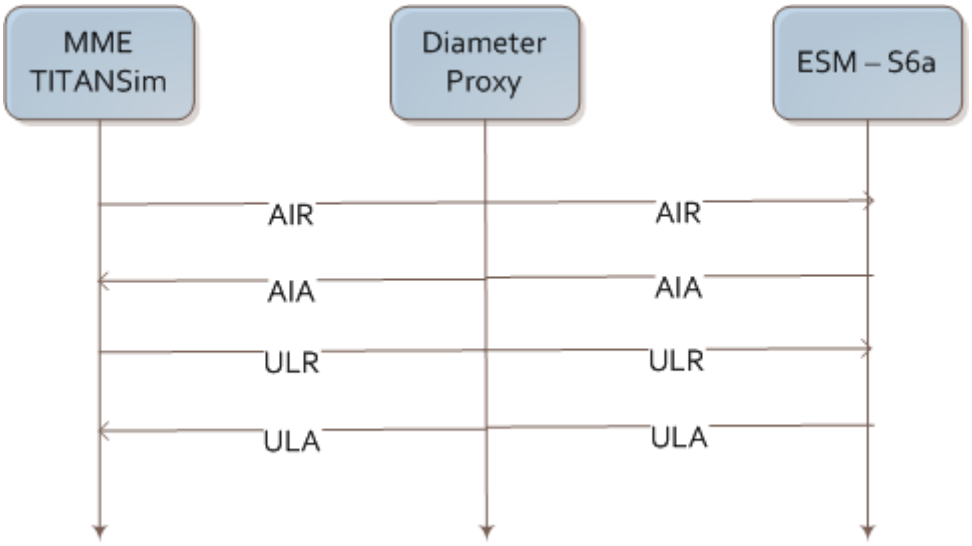


Figure 33: The Initial Attach traffic case

5.4.2 The TITANSim GenApp

As discussed in section 5.2 the TITANSim GenApp software has been chosen to implement the S6a load application on the DS architecture of TITANSim. The TITANSim GenApp is a generic TITANSim traffic generator application that can be used as a test bed for TITANSim development. The TITANSim GenApp is built using the CLL, applibs, TPs, and RT GUI provided by the TITANSim architecture. The GenApp consists of a collection of prewritten LGens for different traffic types such as SIP, Diameter, HTTP, and USSD. Any application developed on the TITANSim GenApp can either use the same prewritten LGen or simply extend and customize it to suit the needs of the application. Since the S6a protocol is based on the Diameter base protocol [14], in this project a new LGen component for S6a traffic type was developed by extending the Diameter base component type. Apart from prewritten LGen types the GenApp also provides a number of standard functions that can be used by all applications for execution control and collecting statistics. Figure 34 below shows the logical run-time view of the TITANSim GenApp. Figure 35 shows the logical run-time view of the MME GenApp created for the S6a load application.

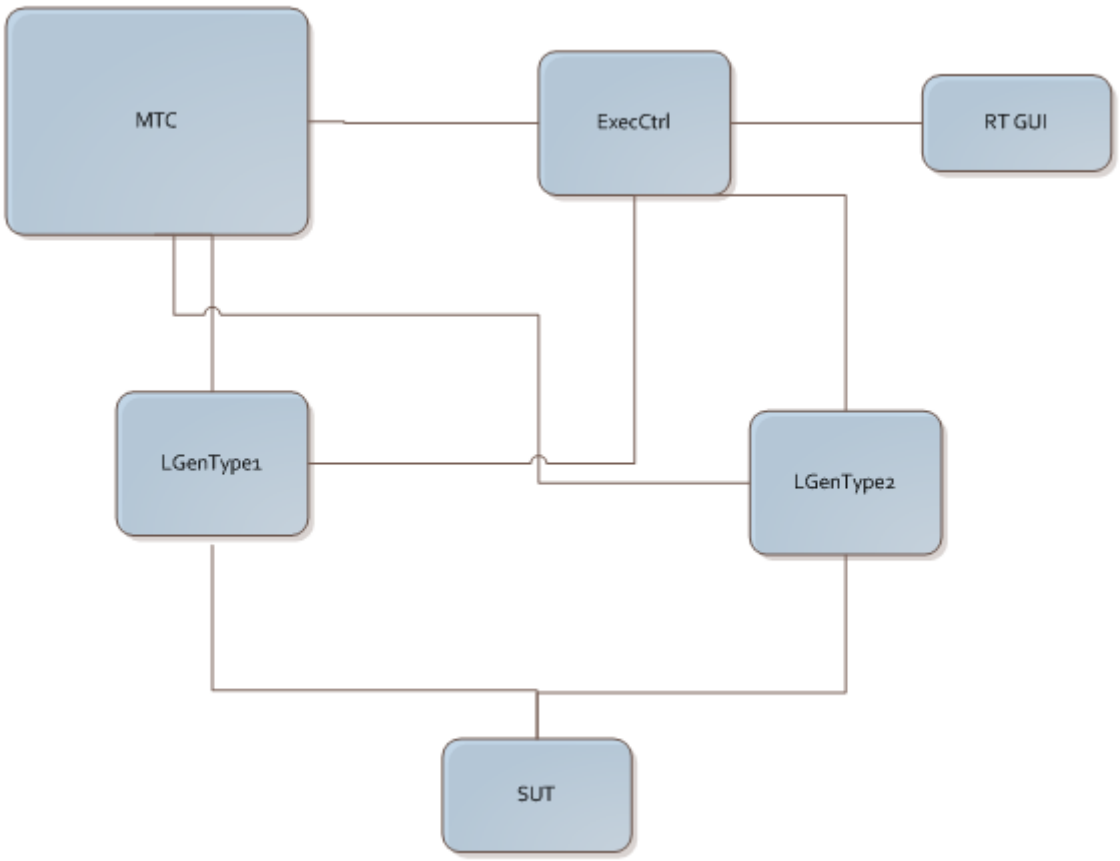


Figure 34: Logical view of GenApp

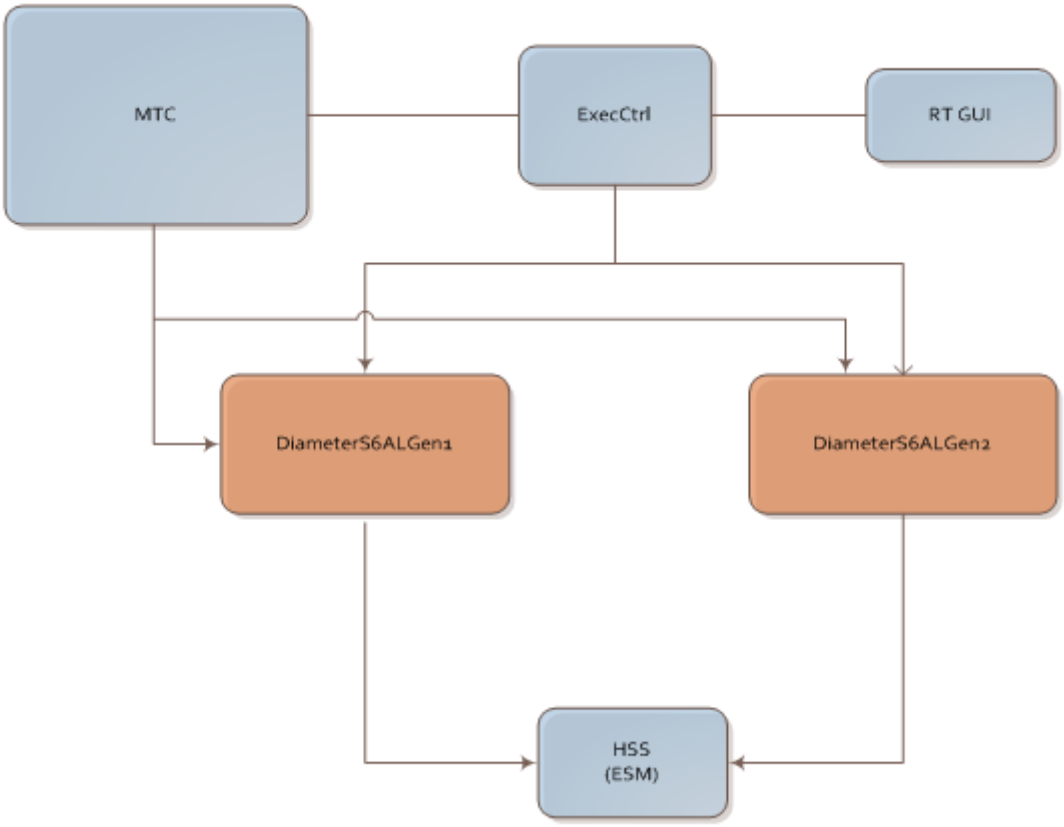


Figure 35: Logical view of MME GenApp

5.4.3 TITANSim framework modifications

The design of the S6a load application required many modifications to the existing standard modules of the TITANSim architecture especially the application libraries for Diameter. These modifications to the TITANSim architecture were performed *before* the actual design of the S6a load application itself. A brief description of these modifications is presented in the following section.

5.4.3.1 Diameter application library

The Diameter application library is one of several protocol libraries that are available in the Application Libraries layer of the TITANSim architecture (refer to section 4.1). This library offers several TTCN-3 modules that contain the basic Diameter AVP type definitions, the FSM of the Diameter base protocol [14], several utility functions to encode/decode AVP's to/from Diameter message, and functions that invoke the transport layer application programming interface (API) (the Test Ports) to send/receive messages.

One of the key modules within the Diameter application library is the Diameter_Types module. This contains the Diameter base protocol AVP definitions and all the other Diameter application protocol AVPs (for example, Cx, and Sh) that are currently supported by the Diameter Application Library. The TITANSim Diameter Application Library initially lacked support for the S6a protocol. Hence, the Diameter_Types module was first extended to support the S6a protocol specific AVPs as per 3GPP TS 29.272 v9.5.0 [21] which is supported by Ericsson's HSS. A Data Definitions File (DDF) was prepared containing the S6a interface specification. This DDF for S6a was used along with DDF files containing the Diameter base specification as input to a tool called DPMG (Diameter Protocol Module Generator). DPMG is an Ericsson proprietary tool utilized by TITANSim to automatically generate and update TTCN-3 modules from DDF files. Thus the Diameter_Types module was updated to provide support for S6a interface with the help of DPMG tool as shown in Figure 36.

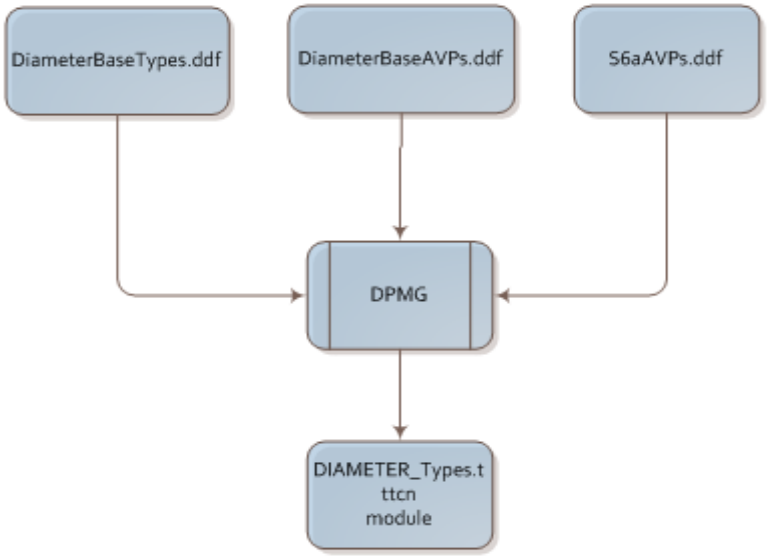


Figure 36: Diameter_Types module generation from the set of DDF files

The library also initially lacked support for the Diameter Initiator[††††] mode, as all the existing Diameter load applications that use this library operate in responder[‡‡‡‡] mode. Thus for this project the Diameter Application Library was updated to support the Initiator mode as per RFC 3588 [14]. A new function was written to initiate a transport connection with the configured Diameter peer. The function uses one of the API functions provided by the IPL4asp socket provided by the Test Ports layer of the TITANSim (refer to section 4.1). Details of this are discussed in the following section.

5.4.3.2 Diameter Test Ports

As discussed in section 4.1 the TP layer of TITANSim offers all the transport functionalities needed for the applications to send/receive messages between the test system and the SUT. The TP layer provides a variety of modules and the appropriate TP module has to be chosen by the application based on its needs. The Diameter based applications developed in TITANSim use IPL4asp test ports.

IPL4asp is a general purpose, session independent test port that provides access to several transport layers over IPv4 and IPv6. The supported transport protocols are TCP, TCP with SSL, SCTP, and UDP over multiple platforms (specifically Linux, Solaris, and Cygwin). For the S6a load application, all the S6a Diameter sessions were run over TCP for the sake of simplicity.

5.4.4 S6a load application design

The design of the S6a load application mainly consists of the design of the S6a LGen component, the FSM that defines the Initial Attach traffic behavior, and other configuration files that contain the values of parameters used by the application at run-time. These entity properties include the number of entities to be used by the application, the AVP values to be sent in the ULR/AIR messages, and socket details such as IP address, port, etc. Figure 37 shows the work flow in the design of the S6a load application.

The S6a LGen component is designed to provide all the capabilities of the LGen base as discussed in section 4.7. In fact this component is an extension of the DiameterBase LGen component provided by the Diameter Application Library, which in turn extends the LGen base component provided by the CLL. The component hierarchy of the S6a load application is shown in Figure 38. The S6a LGen component declares the required entity types, behavior types, and also implements the test steps and events required for the Initial Attach FSM. The declaration of FSM, scenario, and traffic case is done in configuration files that are used by the application at run-time. Thus the load application itself is very flexible and dynamic in behavior as the FSM's are provided at run-time through configuration files.

[††††] In Initiator mode a Diameter peer acts like a client (for example, an MME) and Initiates communication with its peer node.
[‡‡‡‡] In the responder mode a Diameter peer acts like a server (for example, the HSS) and only responds to messages from its peer node and does not initiate the communication.

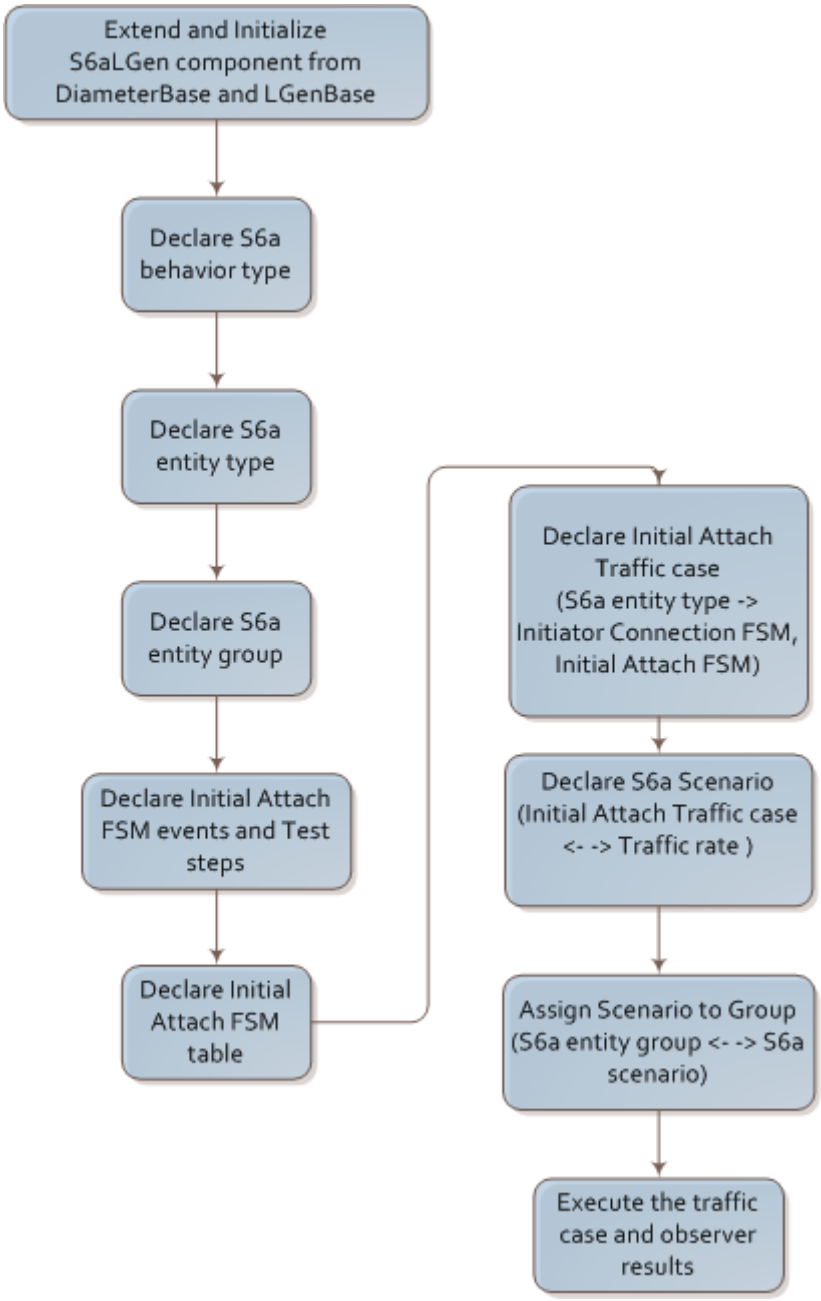


Figure 37: The S6a load application design flow

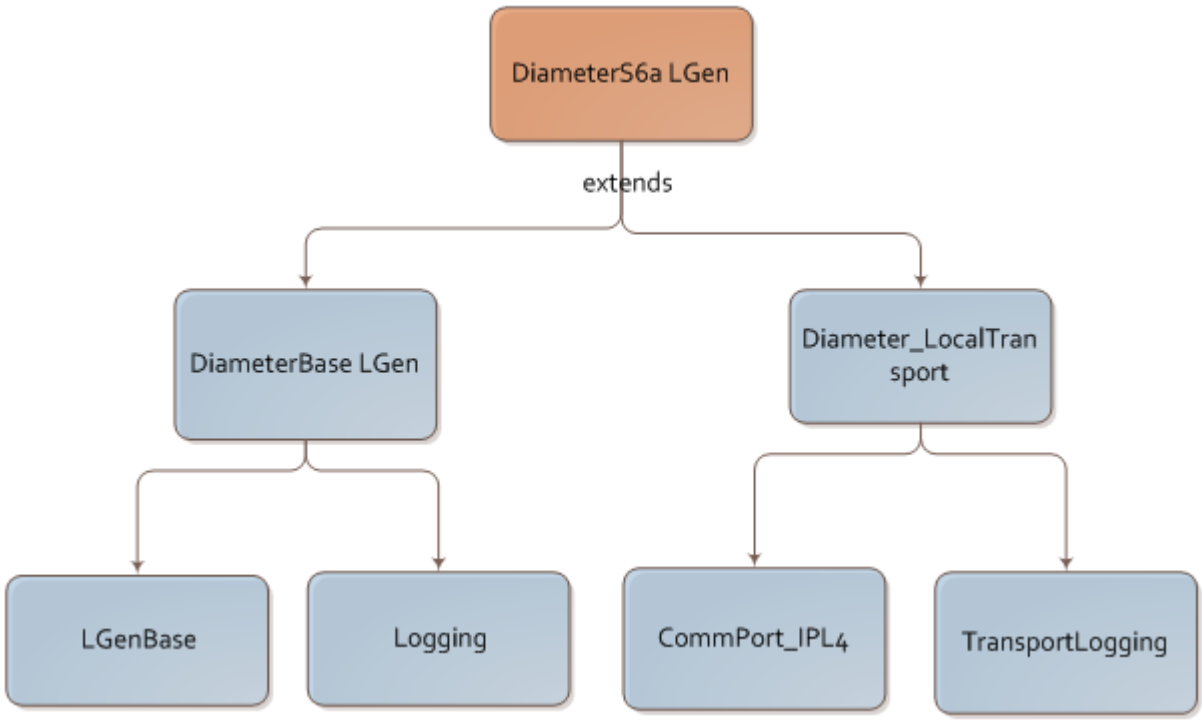


Figure 38: The S6a load application component hierarchy

Two FSMs are designed for the Initial Attach traffic case, namely the Initiator Connection FSM, and the Initial Attach FSM. The Initiator Connection FSM establishes the configured number of transport connections with the Diameter Proxy server. It consists of two states: Idle, and CERWaiting – as shown in Figure 39. The registered events are denoted by 'E' and test step functions by 'A'. The traffic flow of every entity always starts and ends with the state 'Idle'. The LGen base automatically dispatches the 'Start_the_Traffic_case' event to all the S6a entities once the traffic case is started. The entities in the Idle state listen to the 'Start_the_Traffic_case' event and execute the test step function that opens a TCP connection with the configured Diameter peer node –in this case the Diameter Proxy as was discussed in section 5.2.2. If the TCP connection is successfully established, then the S6a LGen component raises an event 'Conn successful'. The entity then executes the test step function to send a CER message to the Diameter Proxy and moves to the next state CERWaiting. If the TCP connection fails, then an event 'Conn failed' is raised upon which the entity reports traffic failure and remains in the Idle state.

When the Diameter Proxy responds with CEA message, the event 'CEA received' is raised and the entity executes the test step function handleCEA to handle the received CEA message. The CEA message is decoded and if it is successful (i.e., contains a DIAMETER_SUCCESS result code AVP), then a 'CEA successful' event is raised and the entity remains in the state CERWaiting after reporting traffic success. If the CEA fails, then a 'CEA failed' event is raised. Then the entity reports traffic failure and remains in the same state. If in the CERWaiting state a 'Stop_the_Traffic_case' event is received (when the tester stops the traffic case execution from the RT GUI), the entity closes the TCP connection with the Diameter Proxy, reports the event 'entity stopped', and moves back to the Idle state.

During traffic execution, if the Diameter Proxy goes down for some reason the TCP connection of each the entity will break, such an event is reported by 'Conn closed' and the entity listens for such an event in the CERWaiting state, reports traffic failure if it occurs, and moves back to the Idle state.

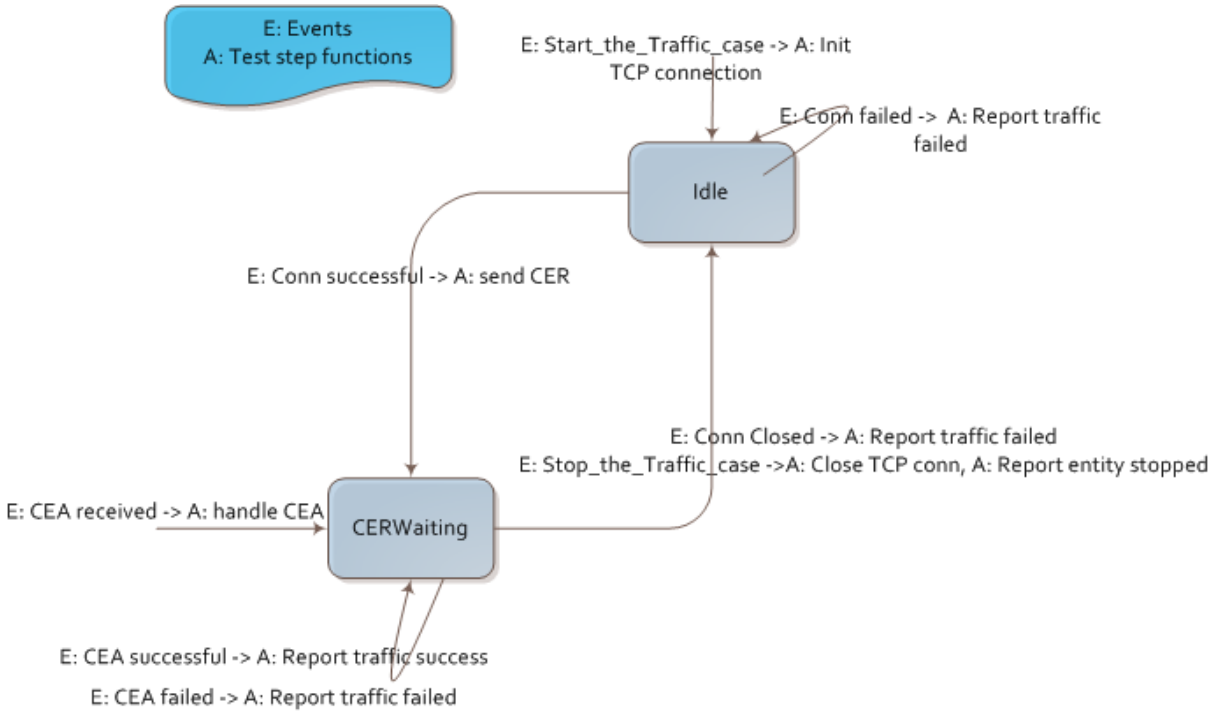


Figure 39: The Initiator Connection FSM

The Initial Attach FSM designed for the traffic case is shown in Figure 40. The FSM consists of three states: Idle, AIRWaiting, and ULRWaiting. The LGen base automatically dispatches the 'Start_the_Traffic_case' event to all the S6a entities once the traffic case is started. The entities in the Idle state listen to the 'Start_the_Traffic_case' event, execute the test step function sendAIR, and move to the AIRWaiting state. In the AIRWaiting state the entity listens to the event 'AIA received', upon reception of which it executes the handleAIA step function to decode the AIA message. If the AIA is successful, then an event 'AIA successful' is raised. It then sends out the ULR message and moves to ULRWaiting state. If the AIA message fails, an 'AIA failed' event is raised. Then the entity reports traffic failure and moves back to the Idle state. Similarly in the ULRWaiting state the entity handles the ULA response message, reports traffic success, and moves back to the Idle state if the ULA is successful. Otherwise it reports traffic failure and moves back to the Idle state if the ULA fails. If in the states AIRWaiting, and ULRWaiting, a 'Stop_the_Traffic_case' event is received (when the tester stops the traffic case execution from the RT GUI) the entity reports the event 'entity stopped' moves back to the Idle state. During traffic execution if the Diameter Proxy goes down for some reason the TCP connection used by the entity would break, such an event is reported by 'Conn closed' and the entity listens for such an event in the AIRWaiting, and ULRWaiting states, reports traffic failure if it occurs, and moves back to the Idle state. The traffic case execution terminates once all the configured entities complete their execution (report either traffic success or

traffic failure). It is also possible to configure the entities to repeat the execution any fixed number of times for a traffic case.

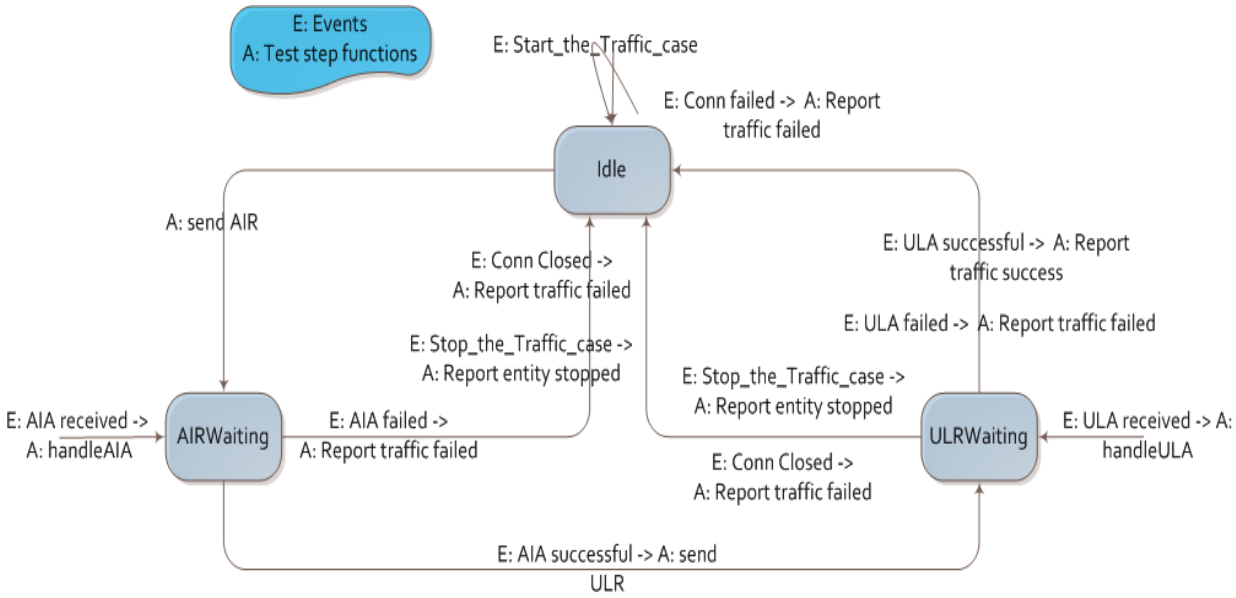


Figure 40: The S6a Initial Attach FSM

As discussed in section 4.1 the test execution is managed with a RT GUI provided by the TITANSim. With this GUI the traffic case can be started, terminated, and various statistics can be viewed (for example, the traffic chart that displays the CPS versus time). Figure 41 shows a screen shot of the RT GUI during the execution of the Initial Attach traffic case. Figure 42 and Figure 43 show the CPS chart displayed by the RT GUI during a sample test execution with 1000 entities and the target CPS set to 50 and 20 CPS respectively.

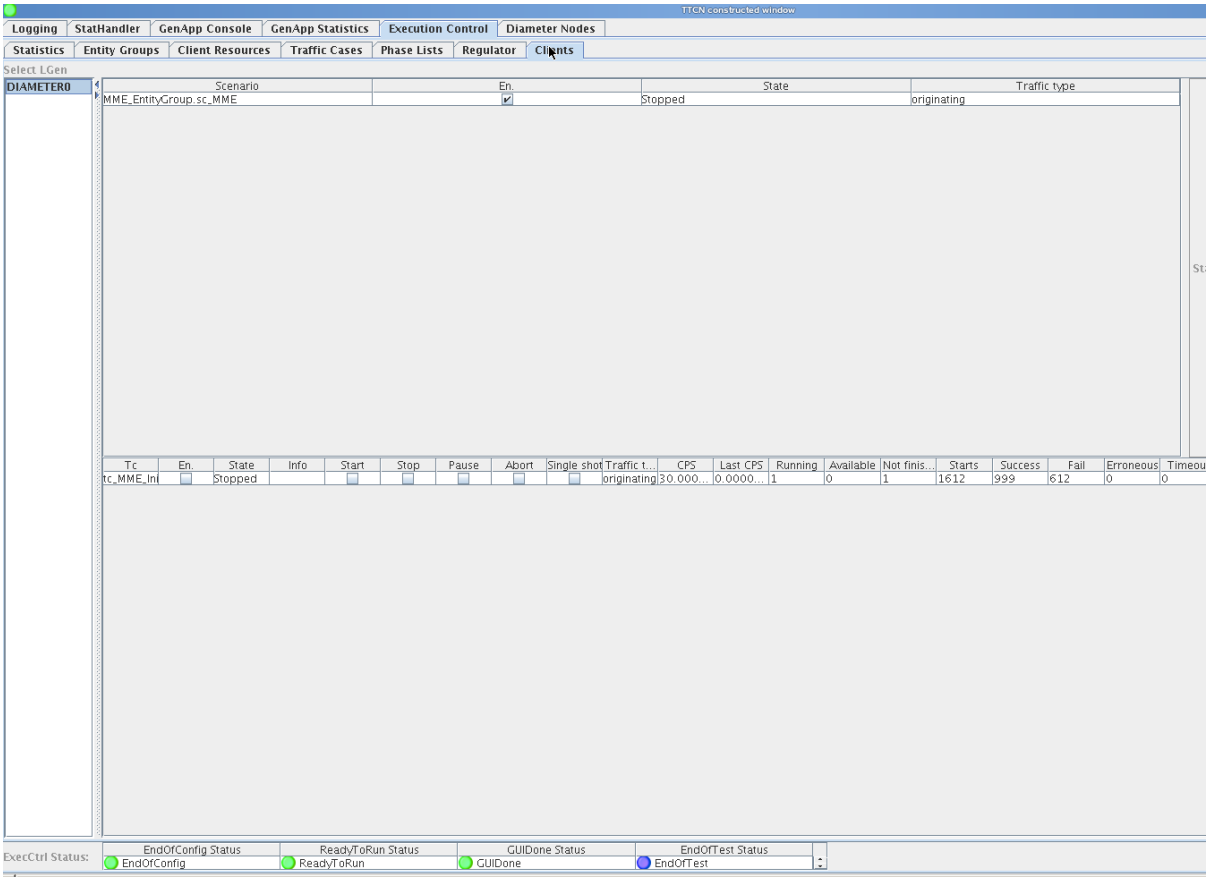


Figure 41: The TITANSim Runtime GUI

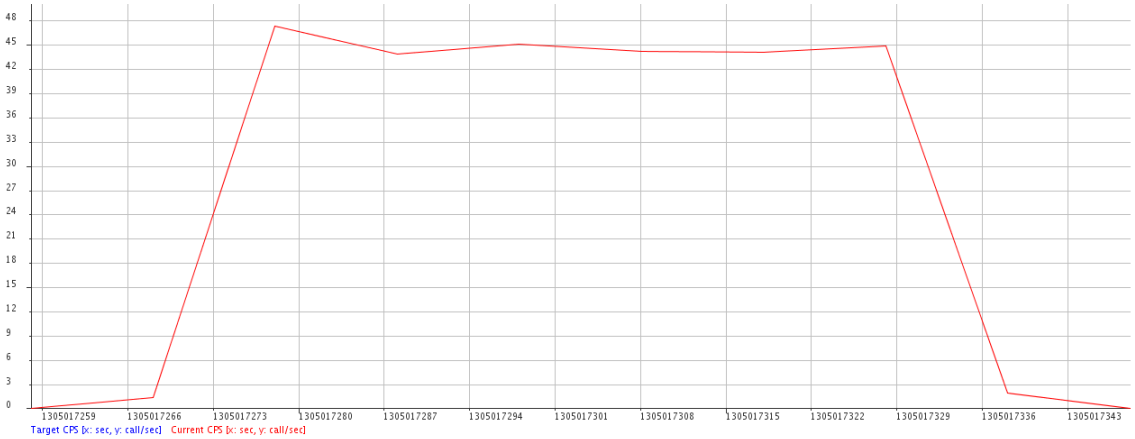


Figure 42: CPS versus Time (seconds) [Target CPS: 50]

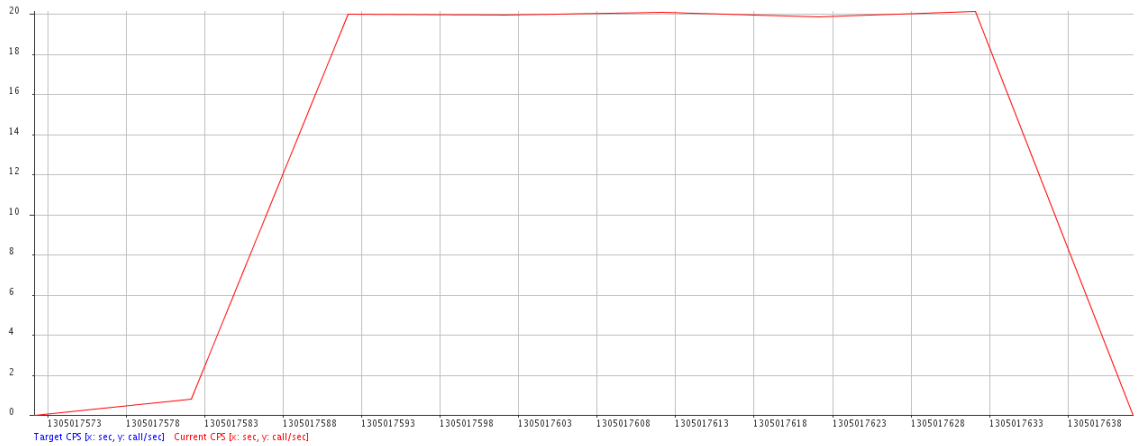


Figure 43: CPS versus Time (seconds) [Target CPS: 20]

5.4.5 Transport mechanism

The TITANSim CLL provides a transport layer that performs many important functions such as message routing (i.e., the routing of incoming and outgoing messages between LGen components), and message buffer management (i.e., the storing of octet-string messages to be sent in a transport connection). The transport layer utilizes the IPL4 or IPL2 test ports provided by the TP layer of TITANSim. It also allows the tester to choose the test port to use for the application either at run-time, or compile-time. It provides three different transport component types that applications can extend to utilize its features.

EPTF_Transport_CT: This component type is test port independent. The type of test port to be used can be specified at run-time through a configuration file.

EPTF_TransportIPL4_CT: This component type uses IPL4asp test ports. It is a simple kernel based solution as shown in Figure 44.

EPTF_TransportIPL2_CT: This component type moves the socket handling to user space and makes it possible to optimize the socket handling independent from the kernel. However it supports only UDP. It is shown in Figure 45.

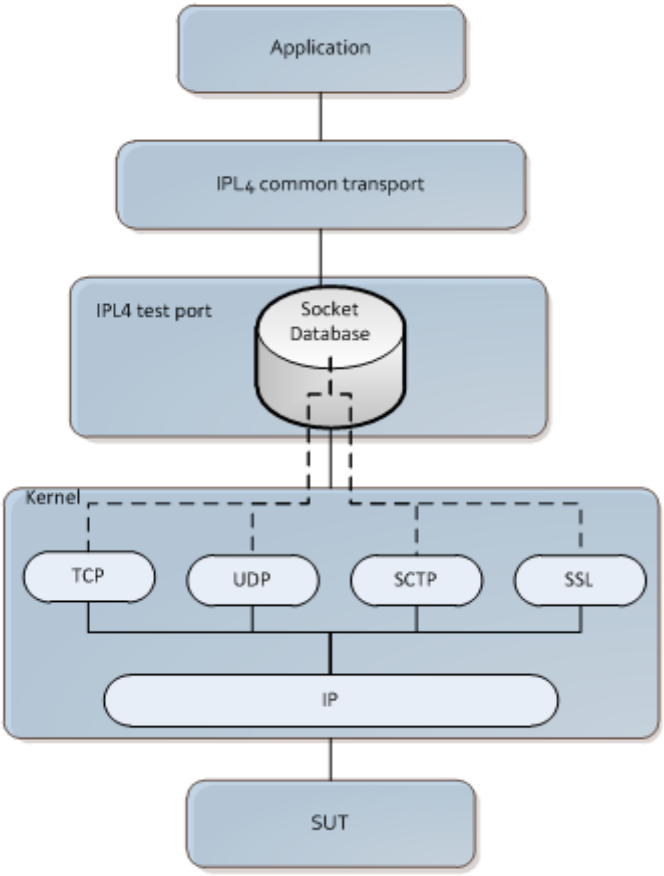


Figure 44: IPL4 transport type

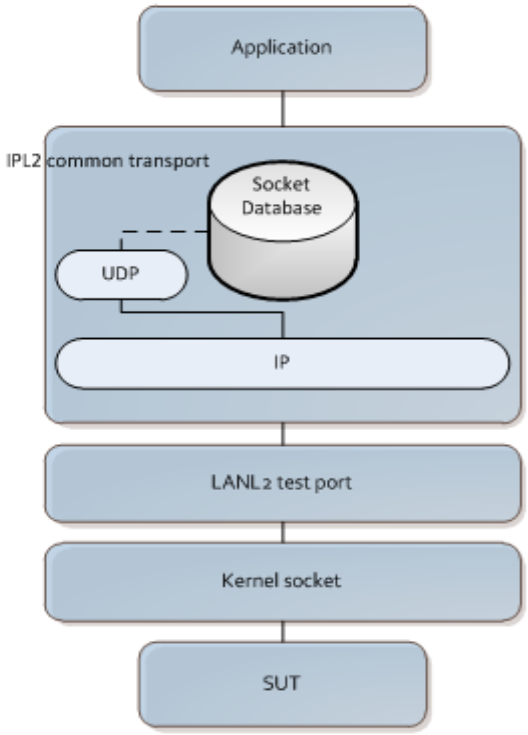


Figure 45: IPL2 transport type

5.4.5.1 Performance comparison of transport types

The IPL4 and IPL2 test ports have their own advantages and disadvantages when it comes to handling sockets, and scalability. IPL2 test port provides better scalability for the transport layer and also performs better than IPL4 when it comes to handling numerous sockets. This is attributed to the fact that in IPL2 test port the socket handling is performed in user space. However, IPL2 test port currently supports only UDP protocol and does not support TCP or SCTP which are required for any Diameter based application. Hence, the S6a load application uses IPL4 test port by extending the *EPTF_TransportIPL4_CT* component. Figure 46 compares the performance of IPL4 and IPL2 when the test system has many IP addresses to open several sockets. Figure 47 compares the performance of IPL4 and IPL2 when the test system uses many ports to open several sockets. In both the cases the performance of IPL2 test remains almost unchanged, but the performance of IPL4 test port steadily declines.

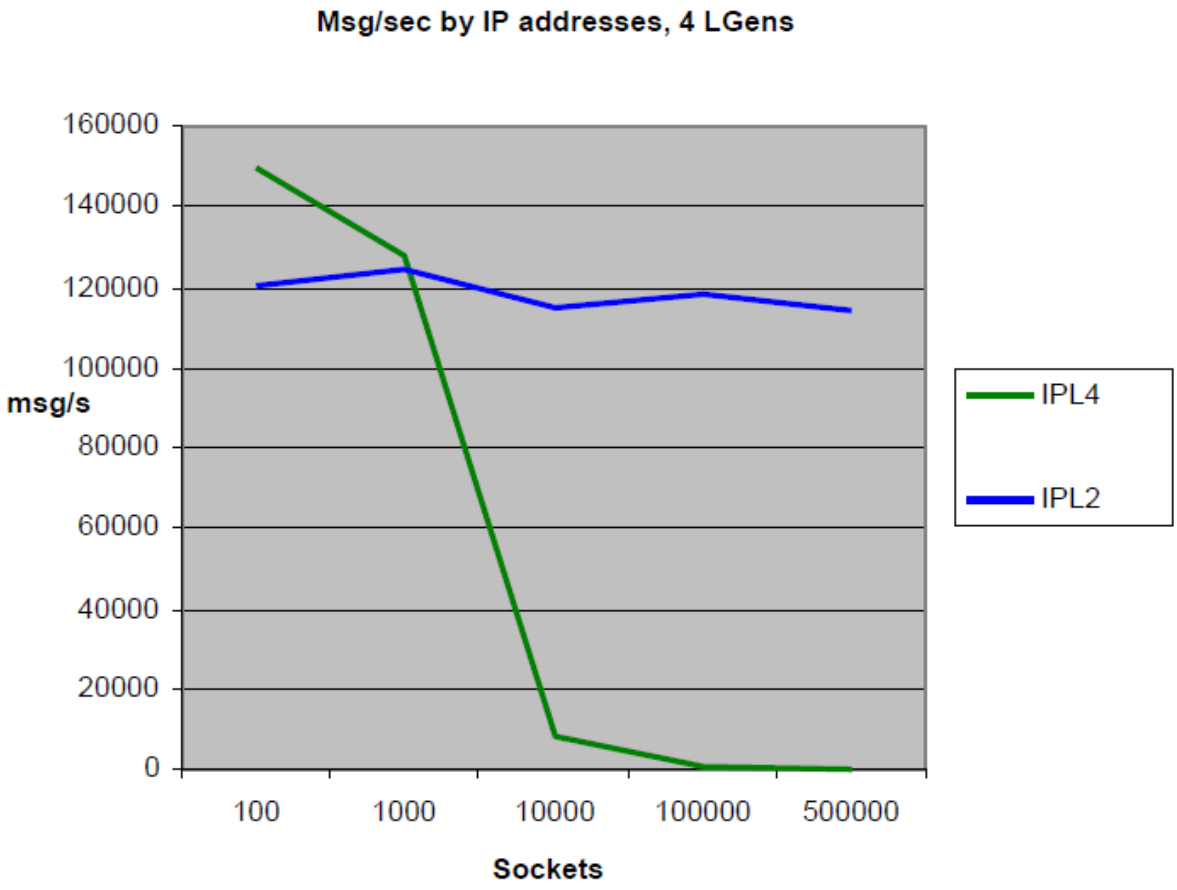


Figure 46[§§§§]: Performance changes while increasing the number of IP addresses

[§§§§] The figure is taken from an Ericsson AB internal document. It appears here with permission from Ericsson AB. It is used here simply to underline the importance of transport multiplexing.

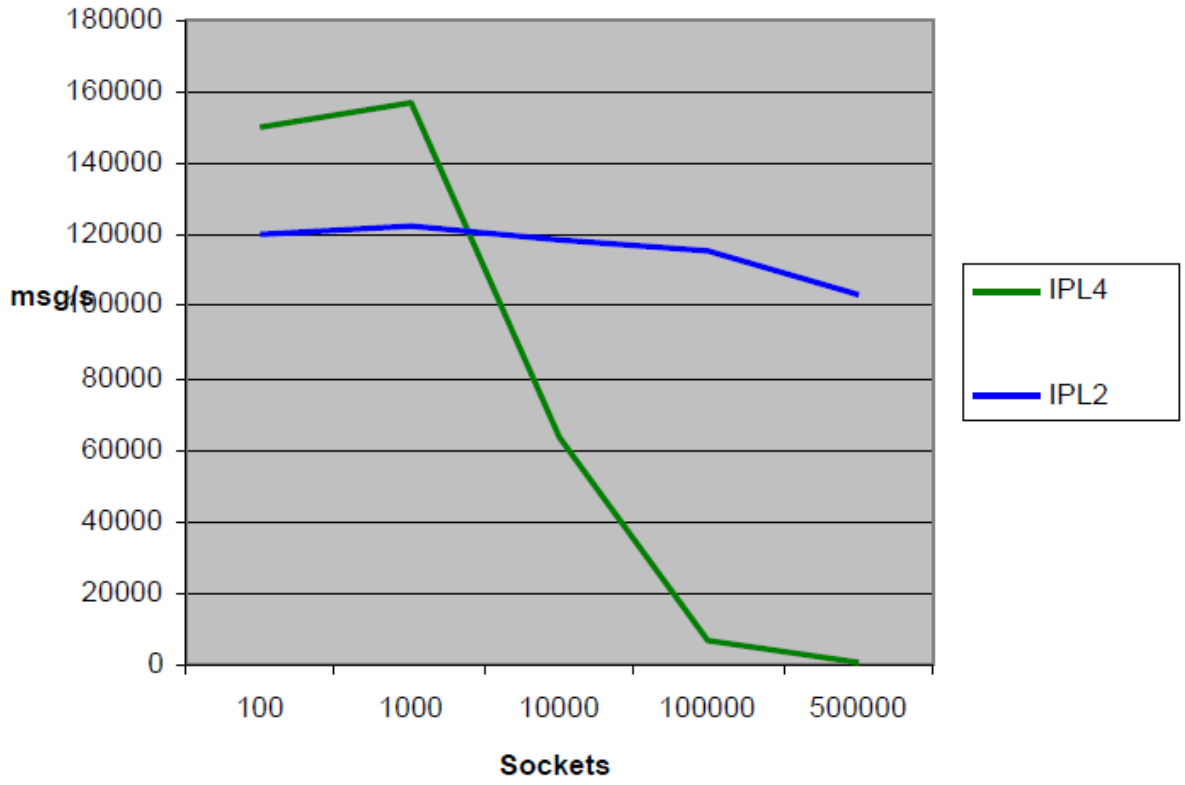


Figure 47[*****]: Performance changes while increasing the number of ports

5.4.5.2 Transport Multiplexing

As discussed in the previous section the IPL4 test port performance declines considerably as more sockets are used by the load application. As a result of this, the message handling capacity of the test system drops and so does the generated load. Thus, the load application has to be designed in such a way that it uses as few sockets as possible to generate the target load.

If each simulated S6a user (i.e., entity) opens a socket towards the Diameter Proxy, there will be about 100 000 sockets used to test the ESM module provisioned with 100 000[†††††] subscribers. This design is not scalable and limits the performance of the IPL4 test ports as seen in Figure 47. Moreover, the Diameter Proxy used in the HSS system test performs optimally only with a maximum of 100 to 200 transport connections.

To solve this problem, the S6a load application is designed to use the transport multiplexing concept, where by multiple simulated entities can be configured to share a single transport connection to send/receive the message to/from the SUT. Thus it is possible to support Initial Attach traffic for around 100 000 ESM subscribers using about 100 TCP connections towards

[*****] The figure is taken from an Ericsson AB internal document. It appears here with permission from Ericsson AB. It is used here simply to underline the importance of transport multiplexing.

[†††††] In Ericsson AB the system test of the ESM module is performed by provisioning a maximum of 100 000 subscribers in the HSS.

the Diameter Proxy. This solution is very scalable and also maintains the performance of the IPL4 test port, and the Diameter Proxy at an optimal level. To implement transport multiplexing, two different entity groups were created – as shown in Figure 48. The first group called the transport group is responsible for establishment and termination of the transport connections to the Diameter Proxy, while the second group called the protocol group is responsible for running the Initial Attach traffic using the transport connections established by the transport group. Each protocol entity is mapped to a transport entity using modulo logic [31]. For example, when the transport group is configured with 100 entities and the protocol group with 100 000 entities, each transport connection created by a transport entity will be shared by 1000 protocol entities.

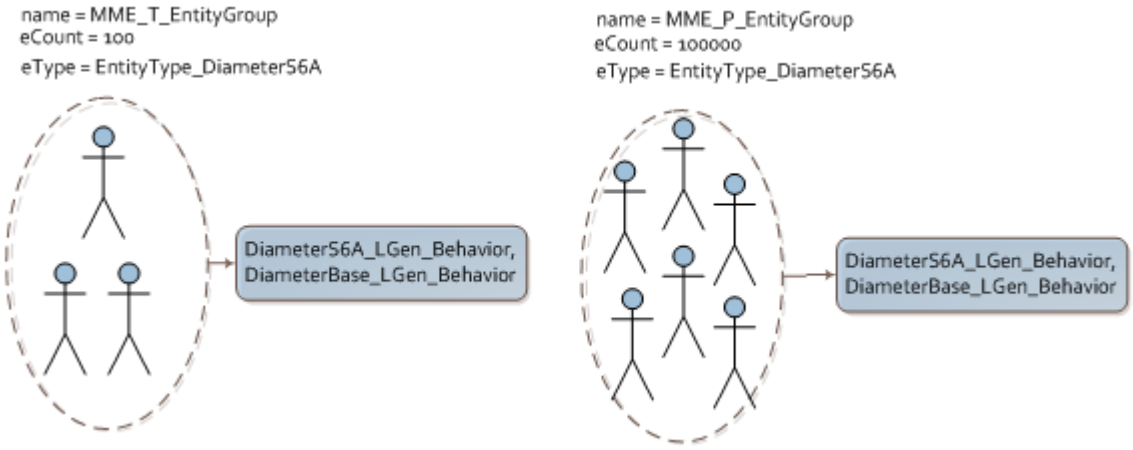


Figure 48: Entity groups for transport multiplexing

5.5 Solution deployment

In the development phase of the S6a load application the tests were performed by installing HSS on a TSP simulator (Maia) as described in section 5.2.3. In order take measurements for performance comparison the S6a load application was *executed in a real system test environment*. This section describes the test hardware and execution architecture used for the tests with their results.

5.5.1 Test hardware

To test the performance of S6a load application and to take measurements a real system test environment was used. The test environment consists of two traffic generator machines (referred to as gteador3 and gteador4) and a TSP cabinet (referred to as eadorm). The test equipment is located at an Ericsson laboratory in Madrid and was accessed remotely from Stockholm. Figure 49 shows an approximate network diagram of the test equipment in the lab.

The TSP hardware is called Network Server Platform (NSP), which is mounted in a cabinet consisting of 4 sub racks. Each sub rack contains a number of processor modules (usually up to a maximum of 12 processors). Figure 50 gives an overview of the node with each sub rack containing processors with duplicate internal Ethernet connections. For this thesis project

NSP 5.0 was used. NSP 5.0 is a powerful, high capacity hardware used for TSP. It consists of a cluster of processor modules. The cluster can contain between 11 and 46 processor modules (depending on the size of the node).

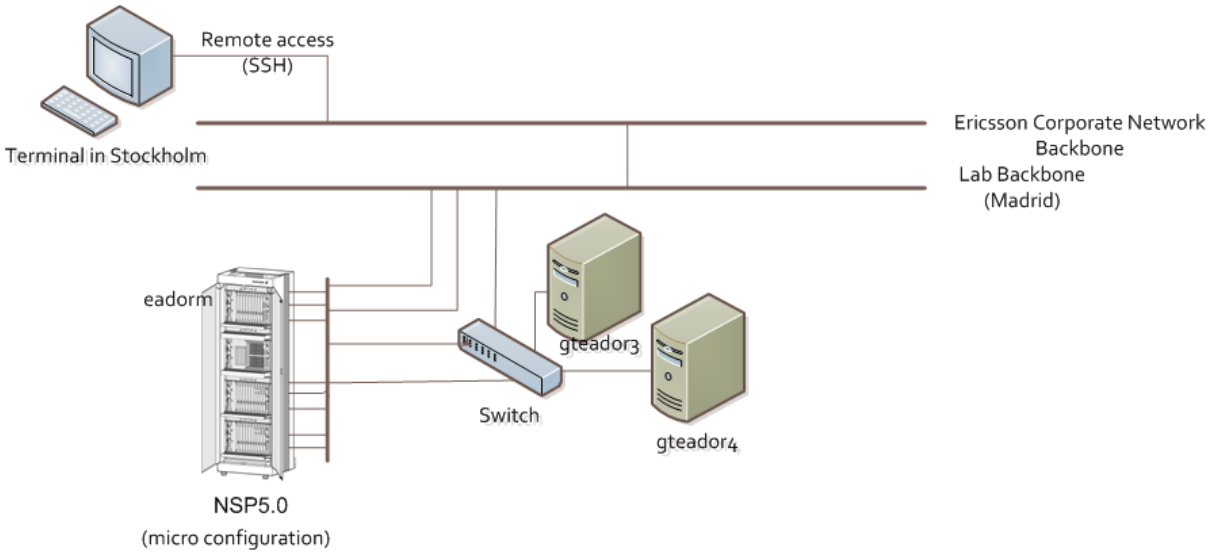


Figure 49: NSP 5.0 cabinet and traffic generators in IP lab

The cabinet eadorm used for this thesis project holds the NSP 5.0 hardware and connected to it is a set of traffic generators called gteador3 and gteador4. TSP 6500 in micro configuration mode (refer to section 5.2.3) was installed in eadorm. After the TSP 6500 installation, the official release of HSS 11b LSV5 software was installed on the node. The TSP and the HSS installation in the cabinet were performed by the TSP configuration management personnel, an expert team in Madrid, Spain. The HSS was provisioned with 100 000 ESM subscribers as described in section 5.2.3.1.

Table 6 shows the configuration of each processor board used in NSP 5.0. There are 8 such processor boards used in the cluster for the TSP in micro configuration.

Table 6: NSP 5.0: Processor and physical memory specification

Low-power Pentium M processor running at 1.8 GHz
2 MB on – die L2 cache
400 MHz Front Side Bus
2 GB DDR RAM

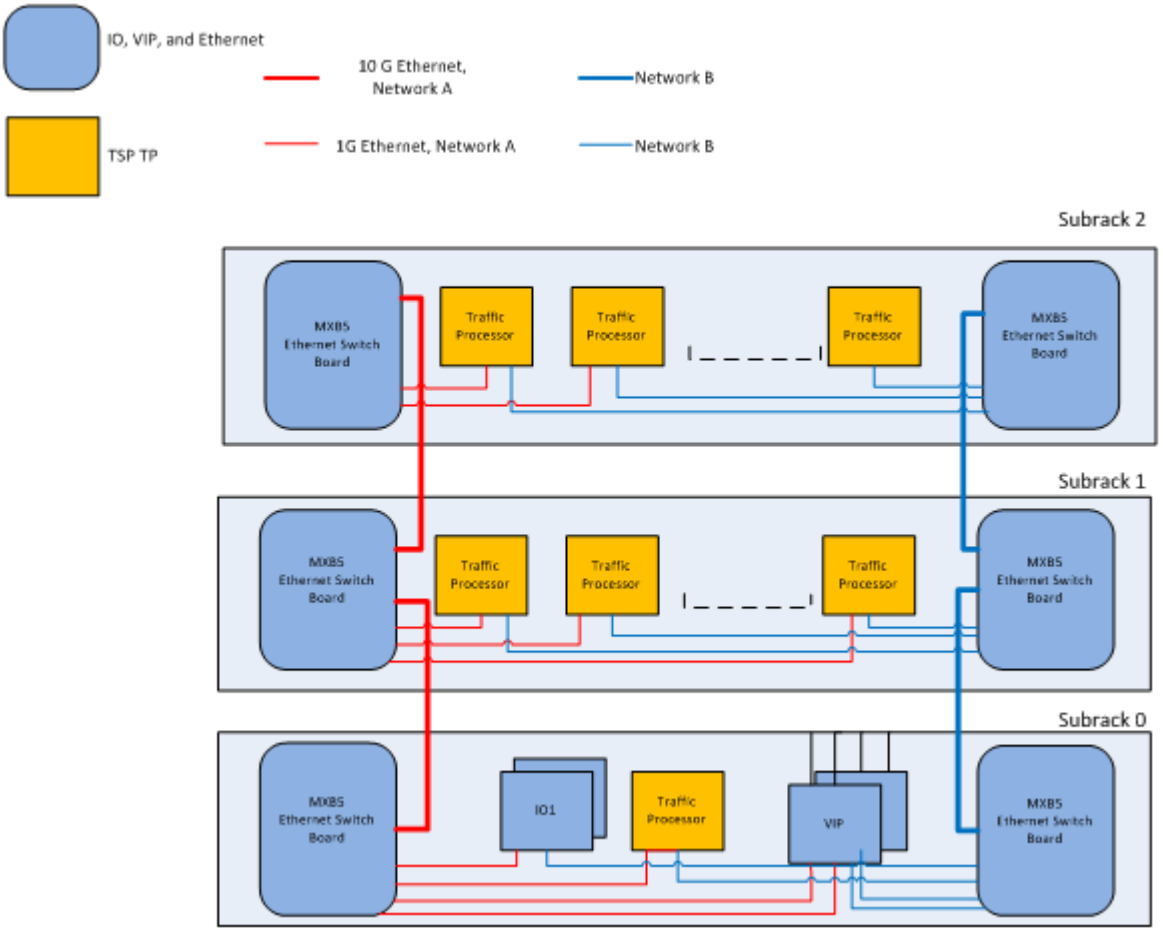


Figure 50: NSP 5.0 hardware overview

The traffic generator machines (gteador3 and gteador4 used for load generation) have the configuration shown in Table 7.

Table 7: Traffic generator: Processor and physical memory specification

No. of processors	**2**
Processor model	**Intel(R) Xeon(TM) CPU 3.20GHz** (single core)
Cache size per processor	**2048 KB**
Total physical memory	**2 GB**

5.5.2 Test execution architecture

As discussed in section 4.1, the S6a load application developed on TITANSim uses the TITAN execution environment. The TITAN execution environment consists of many test components whose behavior is described in the S6a load application. They are basically the MTC and many other PTCs. Apart from these test components the TITAN execution environment also uses other special components which are described in this section.

In TITAN execution environment the test components execute independently, as each of them is a separate operating system process. These components can be run in a single machine or in different machines (distributed test execution). The components communicate with each other using TCP connections via Ericsson proprietary protocols. In TITAN, the test components are classified into three groups based on their function. These groups are main controller, host controller, and test components (further divided into main test component and parallel test components).

5.5.2.1 Main Controller (MC)

The MC is a standalone application provided by the TITAN as a command line interface based control interface for the user to the test executor system. A single instance of MC runs during the entire test execution (irrespective of whether the test execution is running one single machine or distributed of many machines). The MC is manually started by the user. It creates or terminates the MTC upon the user's request. It also shows the test verdict to the user once the test execution is complete.

5.5.2.2 Host Controller (HC)

The HC is an instance (process) of the executable (in this case, the binary of the S6a load application) used for test execution. One instance of HC is started in each machine participating in the test by the user. The HC maintains a TCP connection to the MC for communication, thus during the test execution when the MC wants a new test component to be created on the host (where the HC is running), the HC forks and the resulting child process acts as the new test component.

5.5.2.3 Main Test Component (MTC)

The MTC is also an instance of the test system executable. As discussed in 5.5.2.1, it is the first process that is created on the user's request by the MC. There will be only one instance of the MTC in the entire test system (irrespective of whether there is a single processor or distributed test execution). The MTC starts the test execution by invoking the appropriate control section of the configuration file used for the test. The MTC maintains a TCP connection with the MC for communication.

5.5.2.4 Parallel Test Component (PTC)

For the system test as the desired load cannot be generated by the MTC alone, the test execution utilizes several components executing in parallel to generate the required load. During the test execution the MTC requests the HC to create multiple child processes which are called PTCs. Each PTC is simply another instance of the test system executable. Each PTC maintains a TCP connection with the MC.

5.5.2.5 Distributed execution of S6a load application

As discussed in section 5.5.1, the test hardware consists of two traffic generator machines (gteador3 and gteador4). To fully utilize the available hardware the S6a load application execution was distributed over gteador3 and gteador4. Figure 51 shows the different components (processes) involved in the test and their distribution in the test system.

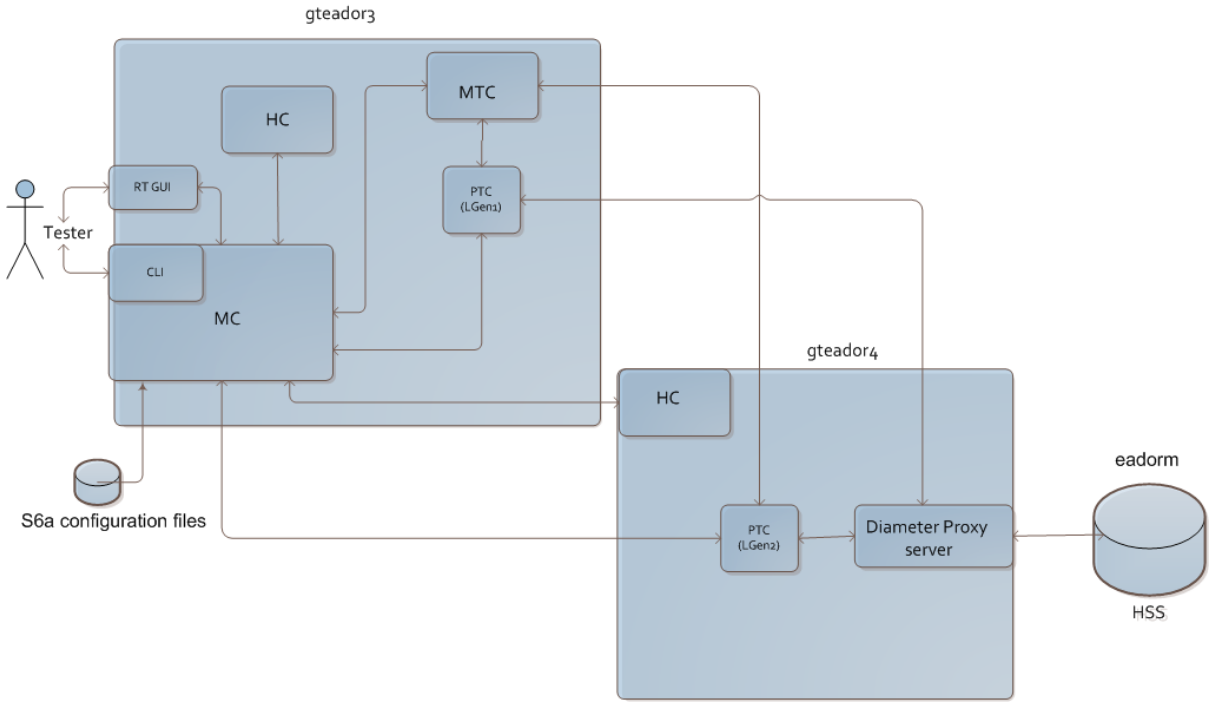


Figure 51[‡‡‡‡‡]: Distributed execution of S6a load application

5.5.3 Test execution and results

The S6a load application was tested by deploying it in different configurations. The following section briefly explains the deployment configuration and the corresponding test results of the S6a load application.

5.5.3.1 Load generation with traffic generators (gteador3, gteador4)

In this test the S6a load application execution was distributed over both the traffic generators, namely gteador3, and gteador4 – as was shown in Figure 51. The application was configured in such a way that only one LGen component was deployed in each traffic generator and each LGen component simulated 15000 ESM subscribers using 50 transport connections. Therefore, the Initial Attach traffic was executed for a total of 30000 ESM subscribers using a total of 100 transport connections towards the Diameter Proxy.

[‡‡‡‡‡] The traffic generators (gteador3 and gteador4) have single core processors. Hence, one LGen was executed in each of the traffic generators. In traffic generators with multiple-core processors, as many LGens can be executed (concurrently) as the number of processor cores (for instance, 2 LGens on a dual-core processor, 4 LGens on a quad-core processor).

The CPU utilization in each of the traffic generator machines remained approximately *75%* as shown in Figure 52 and Figure 53 respectively during the test. The physical memory consumption of the S6a load application in each of the traffic generator machines during the test was approximately *40 megabytes*. The total physical memory utilization in gteador3 and gteador4 is shown in Figure 54 and Figure 55 respectively. Since two traffic generators were used the total CPS was approximately *110* with each traffic generator contributing approximately 55 CPS as shown in Figure 56 and Figure 57. Figure 58 shows a snapshot of the load level (in percentage of the total CPU capacity of each processor) of the 8 processors in the TSP cluster during the test. In Figure 58, m1_s17 and m1_s19 are Linux I/O processors, m1_s13, m1_s15, m1_21, and m1_s23 are DICOS TPs, and m1_s9 and m1_s11 are loader processors. As discussed in section 5.2.3, the real time traffic is handled only by the DICOS TPs. Hence, for determining the average load on the TSP cluster, only the DICOS TPs are considered. In this case, the average load on the DICOS TPs (the HSS) was *17.275%* in the duration of the test. Table 8 shows the maximum, minimum, and average system load, traffic load, and O&M load (in percentage of the total CPU capacity of each processor) for each of the DICOS TPs.

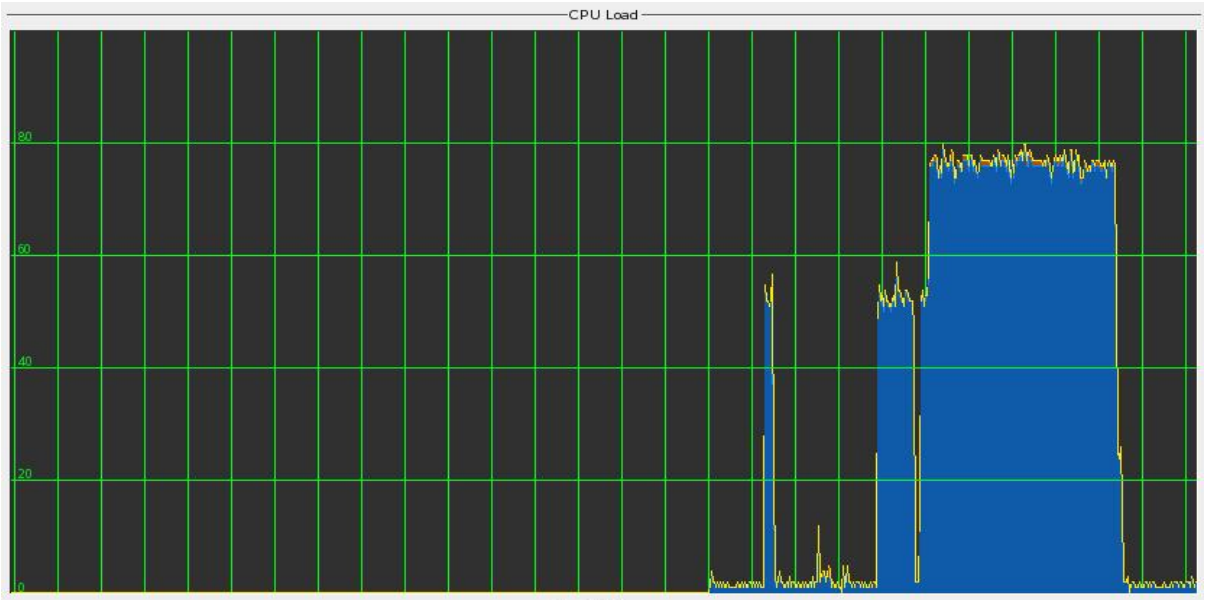


Figure 52: CPU utilization of gteador3

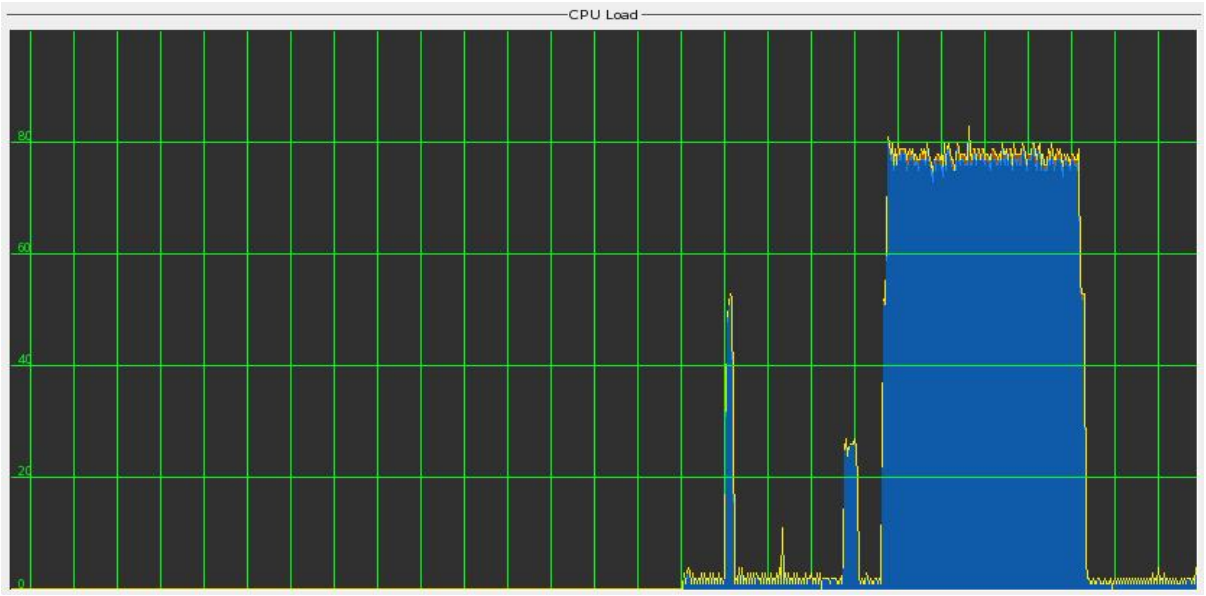


Figure 53: CPU utilization of gteador4

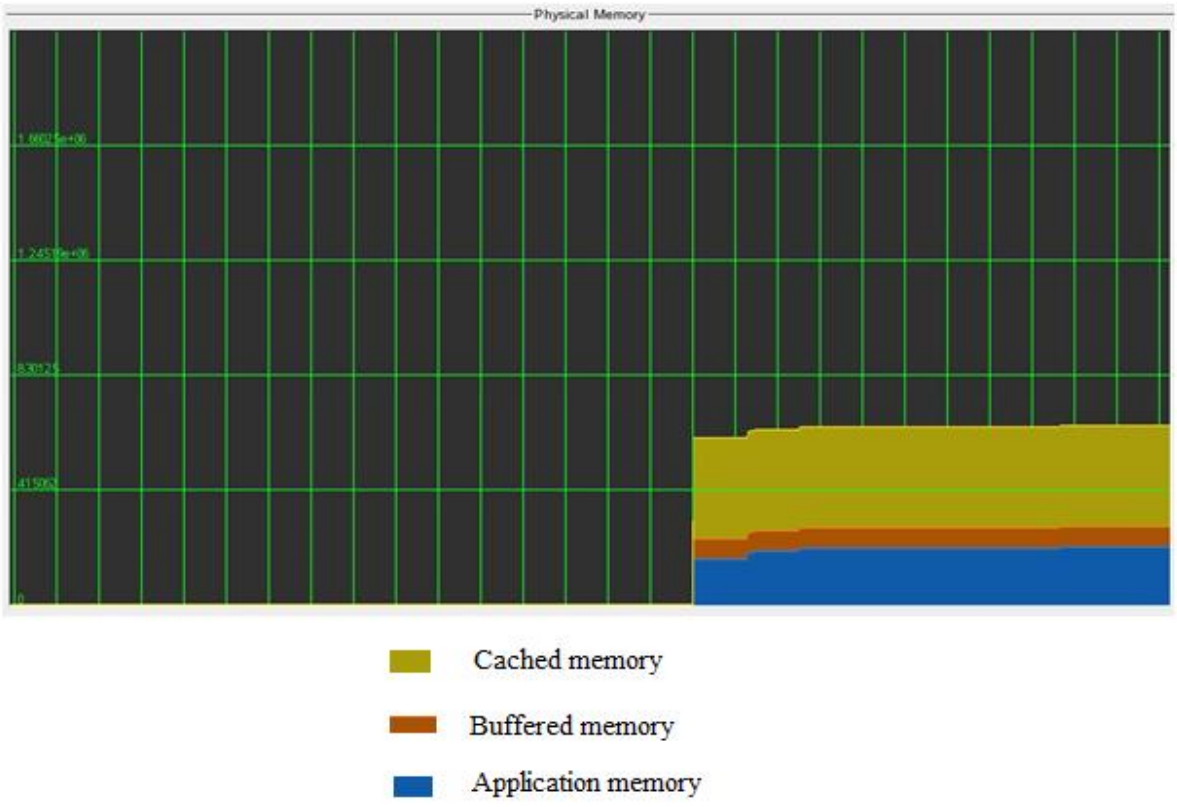


Figure 54: Physical memory utilization in gteador3

Figure 54 shows three colored layers. The top layer indicates the cached memory, the layer in the middle indicates the buffered memory, and the bottom layer indicates the memory consumed by the applications (application memory).

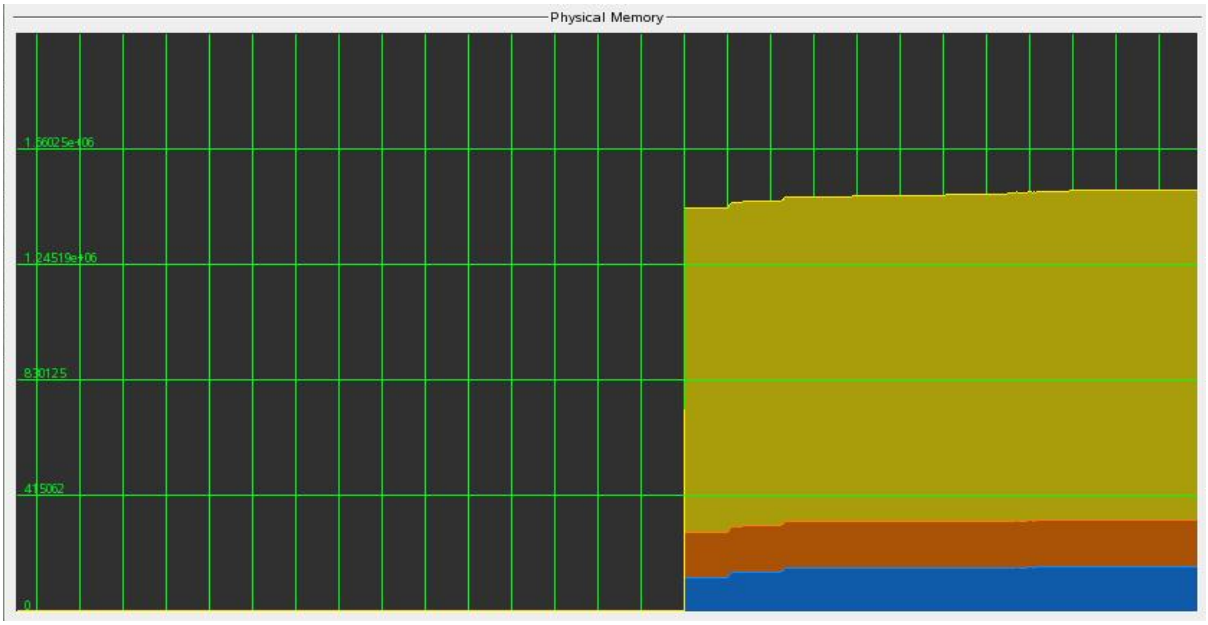


Figure 55[§§§§§]: Physical memory utilization in gteador4

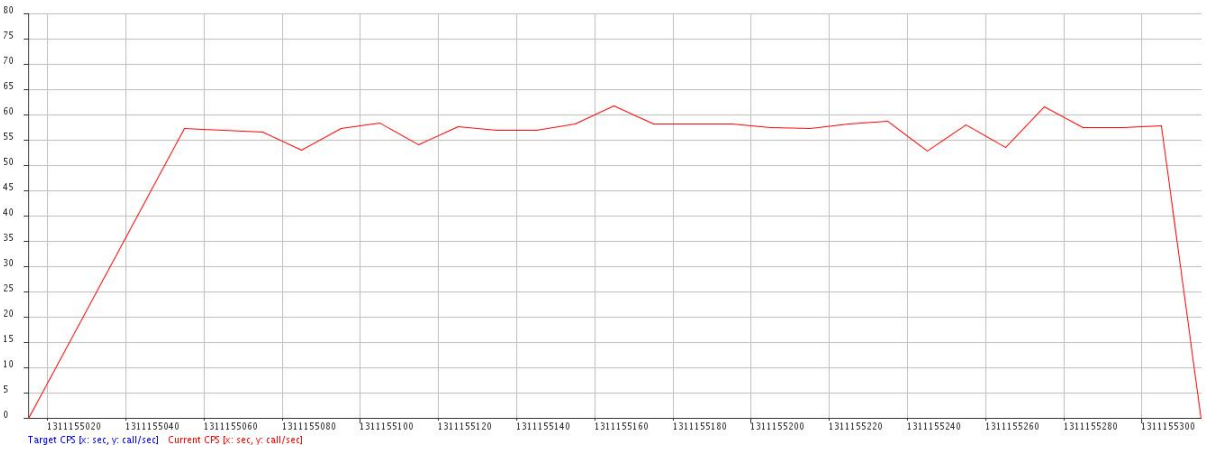
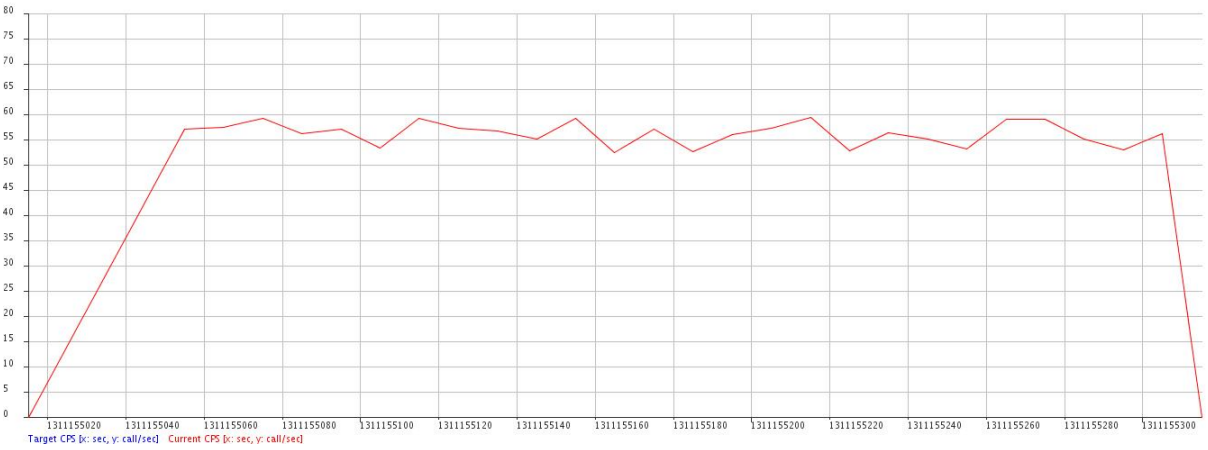


Figure 56: CPS versus Time (seconds) in gteador3



[§§§§§] Figure 55 uses the same color scheme as in Figure 54. The Diameter Proxy process in gteador4 resulted in higher cached memory utilization in gteador4 when compared to gteador3.

Figure 57: CPS versus Time (seconds) in gteador4

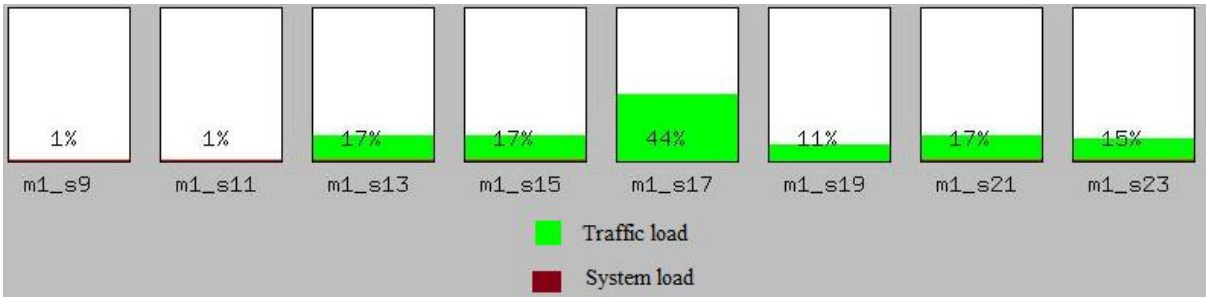


Figure 58[******]: Snapshot of TSP cluster load level in eadorm

DICOS TP/Load %	Proc_m1_s13[††††††]	Proc_m1_s15	Proc_m1_s21	Proc_m1_s23
Max system load	2.0	1.0	1.0	2.0
Min system load	1.0	1.0	1.0	1.0
Average system load	1.996	1.0	1.0	1.012
Max traffic load	32.0	36.0	59.0	32.0
Min traffic load	11.0	11.0	9.0	10.0
Average traffic load	17.461	15.480	15.605	15.312
Max O&M load	2.0	2.0	7.0	2.0
Min O&M load	0.0	0.0	0.0	0.0
Average O&M load	0.055	0.055	0.074	0.051
Total average load	19.512	16.535	16.680	16.375

Table 8: Load distribution in (the HSS) DICOS TPs (in percentage of the total CPU capacity of each DICOS processor)

5.5.3.2 Load generation using high capacity traffic generator

The gteador3 and gteador4 machines are of relatively low capacity, with a single core CPU and a physical memory of only 2 GB each. Hence, the test was repeated using a higher capacity machine called maia15 with the configuration shown in Table 9.

[******] In the figure, m1_s17 and m1_s19 are Linux I/O processors, m1_s13, m1_s15, m1_21, and m1_s23 are DICOS TPs, and m1_s9 and m1_s11 are loader processors

[††††††] The processors in a TSP cluster are identified by the pattern 'Proc_m*_S**'. For instance, Proc_m1_S13 identifies the processor in magazine 1, slot 13 of the NSP cabinet.

Table 9: Traffic generator: Processor and physical memory specification

No. of processors	**4**
Processor model	**Intel(R) Xeon(R) CPU 5160 @ 3.00GHz** (dual core)
Cache size per processor	**4096 KB**
Total physical memory	**16 GB**

In this test the execution was carried out on a single machine (maia15). Thus, all the test components and the Diameter Proxy were running in the same machine. Two LGen components were deployed with each LGen simulating 15 000 ESM subscribers using 50 transport connections with the Diameter Proxy. Therefore the test covered a total of 30 000 ESM subscribers using a total of 100 transport connection with the Diameter Proxy.

Figure 59: CPU utilization of maia15

The CPU utilization increased from *3%* (at the start of the test) and remained approximately at *68%* (the third broad peak in Figure 59) during the test. The physical memory consumption of the S6a load application was *63 megabytes* during the test. Figure 61 shows the total physical memory utilization of maia15 during the test. The total CPS generated by the traffic generator during the test was approximately *320 CPS* (each LGen contributing approximately *160 CPS*). Figure 60 shows the CPS versus Time chart of one of the LGens during the test.

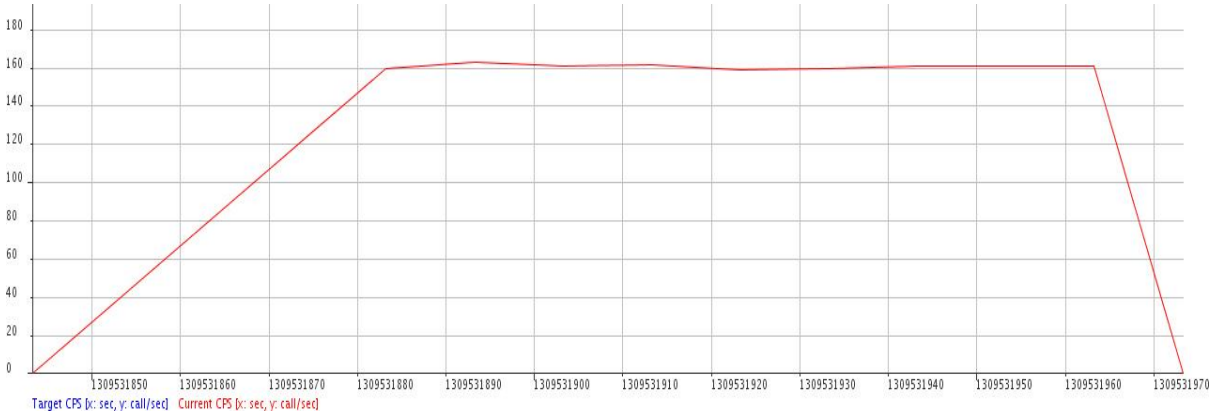


Figure 60: CPS versus Time (seconds) of maia15

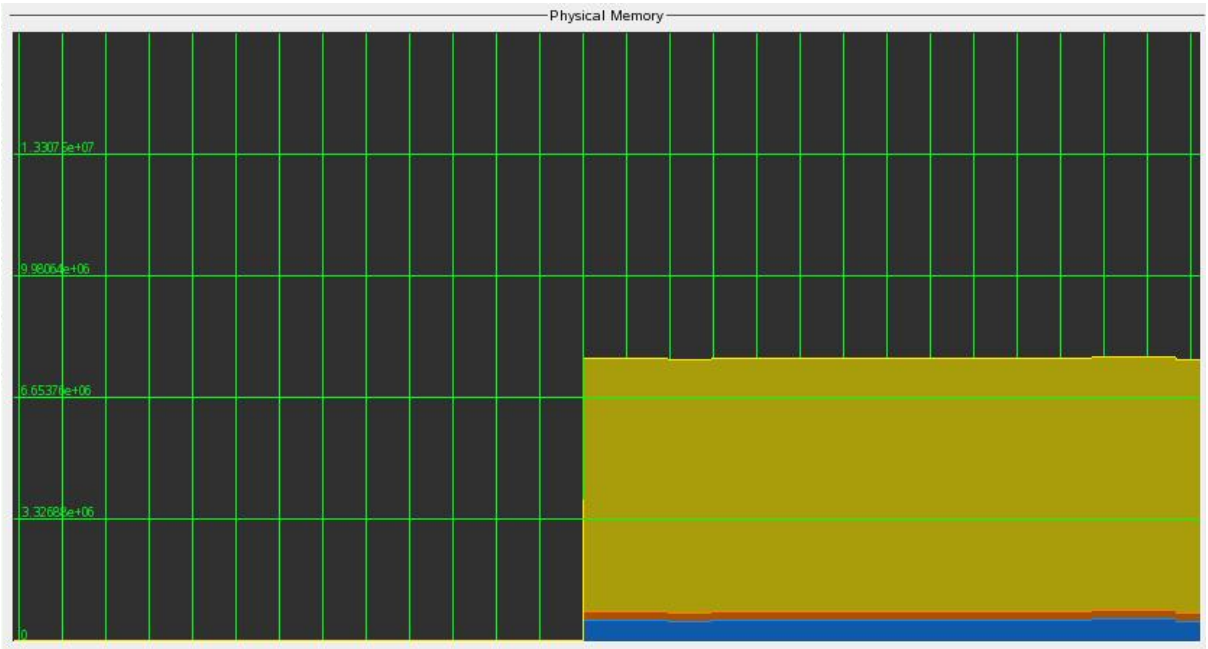


Figure 61[‡‡‡‡‡‡]: Physical memory utilization of maia15

Figure 62 shows a snapshot of the load level (in percentage of the total CPU capacity of each processor) of the 8 processors in the TSP cluster during the test. Table 10 shows the maximum, minimum, and average system load, traffic load, and O&M load for each of the DICOS TPs. The average load on the DICOS TPs (the HSS) was *48.366%* in the duration of the test.

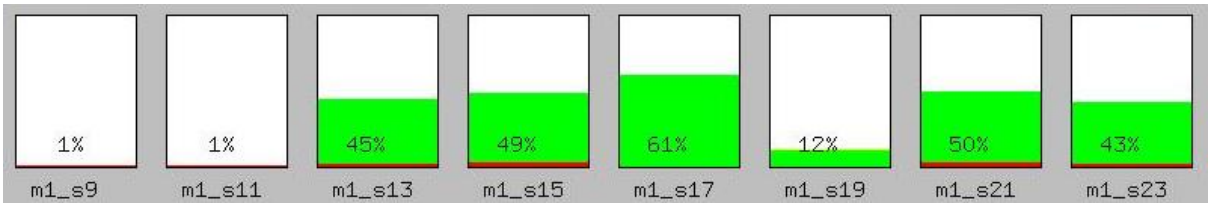


Figure 62[§§§§§§]: Snapshot of TSP cluster load level in eadorm

[‡‡‡‡‡‡] Figure 61 uses the same color scheme as in Figure 54

DICOS TP/Load %	Proc_m1_s13	Proc_m1_s15	Proc_m1_s21	Proc_m1_s23
Max system load	2.0	3.0	4.0	2.0
Min system load	2.0	2.0	3.0	2.0
Average system load	2.0	2.986	3.261	2.0
Max traffic load	57.0	57.0	78.0	58.0
Min traffic load	33.0	39.0	40.0	36.0
Average traffic load	43.507	48.043	47.884	43.493
Max O&M load	2.0	3.0	5.0	2.0
Min O&M load	0.0	0.0	0.0	0.0
Average O&M load	0.058	0.072	0.101	0.058
Total average load	45.565	51.101	51.246	45.551

Table 10: Load distribution in (the HSS) DICOS TPs (in percentage of the total CPU capacity of each DICOS processor)

§§§§§§ Figure 62 uses the same color scheme as in Figure 58

6 Analysis

In this chapter, the results of similar tests (the test described in section 5.5.3.1) performed with the load application based on CS architecture of TITANSim are presented. Then a thorough analysis is performed by comparing it with the results from section 5.5.3.1 (results of test with S6a load application).

6.1 BAT load application

The CS based load application which is currently used for system test at HSS in Ericsson is called the Background Activities Test (BAT). The BAT load application is used to test various modules within the HSS such as ESM, IP Subscription Manager, Subscriber Data Access, and Wireless Subscription Module. The BAT load application for ESM uses a separate set of configuration files for the test deployment. However, for the sake of simplicity details of the *BAT environment set up and its configuration* is *not given*. The test environment is exactly the same as described in section 5.5.1, but in this case the *BAT load application is executed* in the traffic generators instead of the S6a load application.

The BAT load application for ESM uses a mix of traffic cases for load generation. For example, a scenario can be configured to run a traffic mix of Initial Attach, HSS initiated Detach, Insert Subscriber Data, Handover LTE to 3G, and many more. It is also possible to configure the percentage contribution of each of these traffic cases to the total load. For example, a scenario can be configured to generate 10% Insert Subscriber Data traffic, 30% Handover traffic, 20% Purge Subscriber traffic, and 40% Service Authentication traffic. However for this thesis project, we consider only the Initial Attach traffic case. Therefore, for this test only the Initial Attach traffic was generated (i.e., the Initial Attach traffic constituted 100% of the traffic).

6.2 BAT results

The BAT load application for ESM was configured to run the Initial Attach traffic for 30 000 ESM subscribers using 200 LGen components (PTCs), and the test execution was distributed over gteador3 and gteador4 (100 LGen components in each traffic generator) as shown in Figure 51. Each LGen component uses a separate transport connection to the Diameter Proxy. Therefore, the BAT load application uses a total of 200 transport connections to run the Initial Attach traffic for 30 000 ESM subscribers.

The CPU utilization largely remained within the range of *3% to 10%* with occasional peaks up to *25%* during load generation in gteador3 as can be seen in Figure 63. Figure 64 shows the CPU utilization of gteador4 which largely remained within the range of *3% to 5%*. The physical memory consumption of the BAT load application during the test was approximately *154 megabytes* in gteador3 and *145 megabytes* in gteador4. Figure 65 and Figure 66 show the total physical memory utilization in gteador3 and gteador4 respectively, during the test. The

CPS chart is shown in Figure 68. The CPS varied from a *minimum of 60 to a maximum of 385* during the test.

Figure 67 is a snapshot of the load level (in percentage of the total CPU capacity of each processor) of the 8 processors in the TSP cluster during the test. Table 11 shows the maximum, minimum, and average system load, traffic load, and O&M load for each of the DICOS TPs. The average load on the DICOS TPs (the HSS) was *5.576%* in the duration of the test.

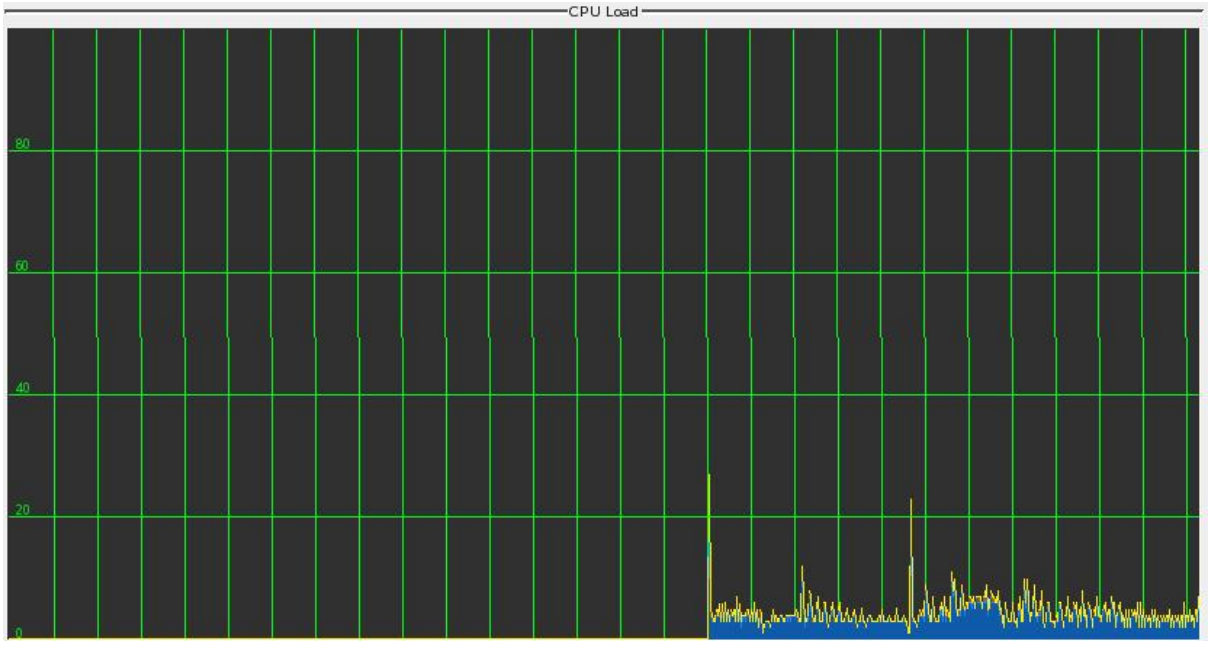


Figure 63: CPU utilization of gteador3

Figure 64: CPU utilization of gteador4

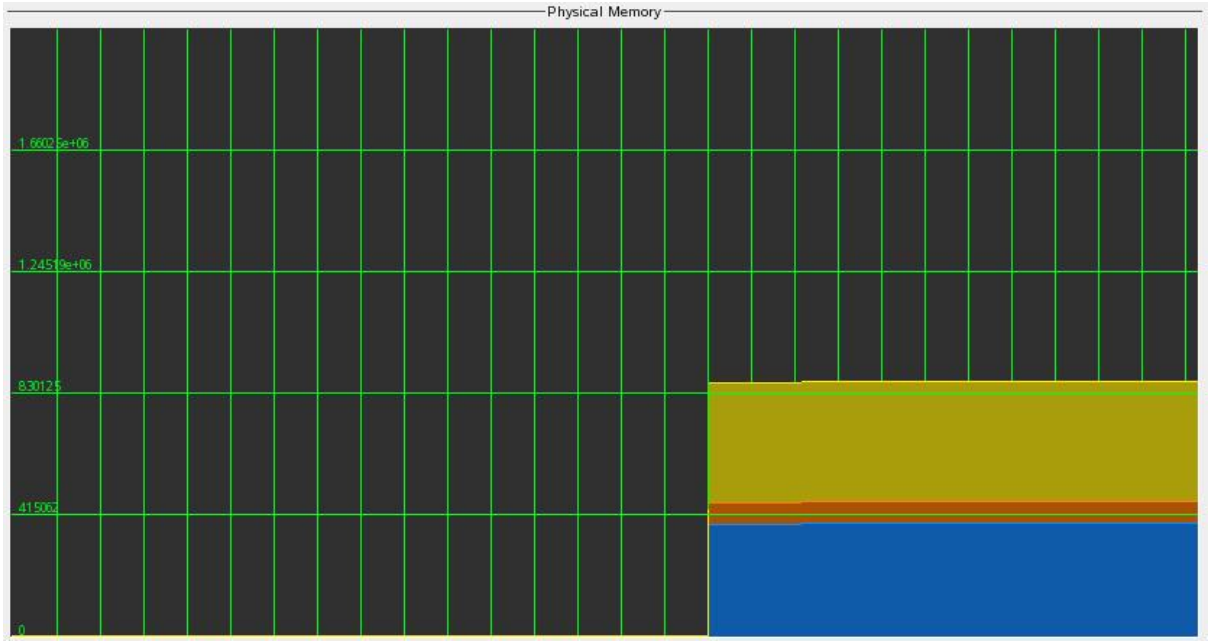


Figure 65[*******]: Physical memory utilization of gteador3

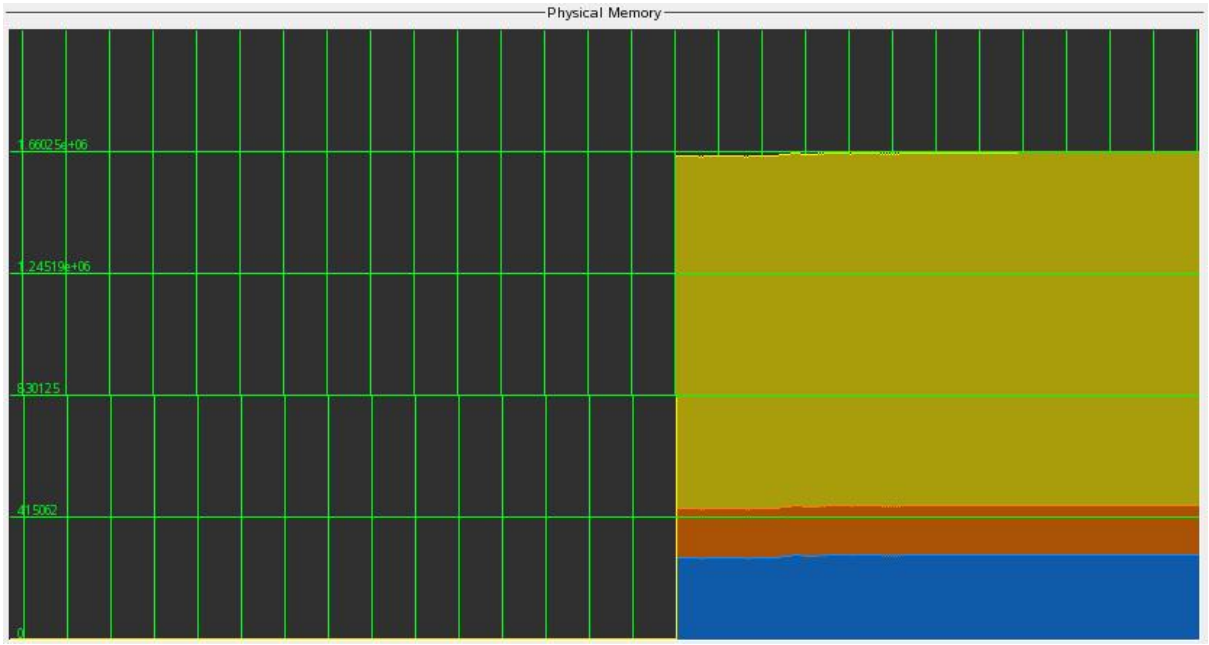


Figure 66[†††††††]: Physical memory utilization of gteador4

[*******] Figure 65 uses the same color scheme as in Figure 54
[†††††††] Figure 66 uses the same color scheme as in Figure 54

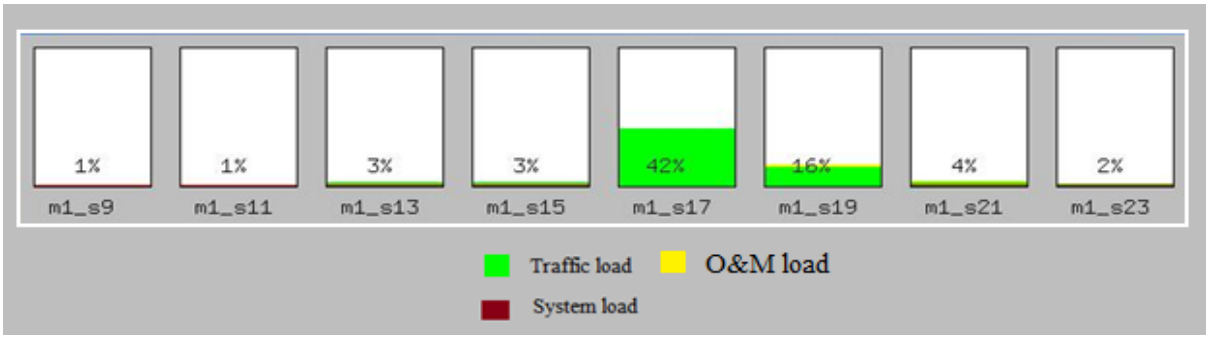


Figure 67: Snapshot of TSP cluster load level in eadorm

Figure 68: CPS versus Time (seconds) of BAT application

DICOS TP/Load %	Proc_m1_s13	Proc_m1_s15	Proc_m1_s21	Proc_m1_s23
Max system load	1.0	1.0	1.0	1.0
Min system load	1.0	1.0	1.0	1.0
Average system load	1.0	1.0	1.0	1.0
Max traffic load	22.0	22.0	56.0	22.0
Min traffic load	0.0	0.0	0.0	0.0
Average traffic load	3.465	2.915	3.317	2.979
Max O&M load	5.0	4.0	7.0	5.0
Min O&M load	0.0	0.0	0.0	0.0

Average O&M load	1.444	1.317	1.408	1.458
Total average load	5.908	5.232	5.725	5.437

Table 11: Load distribution in (the HSS) DICOS TPs (in percentage of the total CPU capacity of each DICOS processor)

6.3 Comparison of results

A comparison of the test results of the BAT load application and the S6a load application will be made in terms of the test system resource consumption, load generation capacity, and other factors such as the test execution time, and CPU utilization of the traffic generators. The results presented in this section are based on the tests described in section 5.5.3.1 and 6.2. Table 12 compares the test resource consumption of the BAT and S6a load application during the test. The total physical memory and the number of processes used by the application during the test are shown considering both the traffic generators. The test execution time is basically the time taken to execute the traffic case (i.e., to run the Initial Attach procedure for 30 000 ESM subscribers).

6.3.1 Test system resource consumption

It is evident from Table 12 that the BAT load application is very resource intensive. It consumes more physical memory and operating system processes compared to the S6a load application. The BAT load application consumes more than thrice the physical memory consumed by the S6a load application for generating the same amount of traffic (i.e., generate Initial Attach traffic for 30 000 ESM subscribers). The physical memory consumption of the BAT load application would increase by several times when tests are run with a mix of traffic cases for higher number of ESM subscribers (say 100 000).

The BAT load application executes the traffic case in three phases namely, pre-execution phase, load phase, and post-execution phase. In the pre-execution phase existing logging information is deleted, O&M events such as alarms are verified in the HSS, etc. In the load phase the Initial Attach traffic is generated. In the post-execution phase, a Detach procedure is carried out for the subscribers (for which the Attach procedure was carried out in the load phase). The time taken for the execution of each of these phases is shown in Table 12. Since the S6a load application does not run the Detach procedure, the time taken for post-execution phase is ignored in the discussion. Considering only the time taken for the load phase execution (when the Initial Attach traffic was generated), we can say that the BAT load application generates traffic faster than the S6a load application (the load phase execution of the Initial Attach traffic case in BAT load application completed in 223 seconds, where as the execution of Initial Attach traffic case in S6a load application completed in 282 seconds). There are many factors that lead to this difference, such as the inherent architectural difference in scheduling, and differences in the configuration of logging operations of the two applications during the test. However, the time taken for the overall test execution of the BAT load application (381 seconds) is higher than the S6a load application (282 seconds).

The execution time of generating the traffic cases in the S6a load application can be improved further by optimizing the FSMs. The FSMs can be made simpler by optimizing the events and test step functions that entities execute to generate the traffic. It is clear from Table 12 that the S6a load application results in higher CPU utilization when compared to the BAT load application. Thus, the S6a load application makes efficient use of test system resources to generate a consistently high load when compared to the BAT load application (which only achieved the target load near the end of the test).

Table 12: CS/BAT versus DS/S6a test system resource consumption

Parameter	CS/BAT result	DS/S6a result
Total physical memory used (megabytes)	299	80.4
Total no. of processes‡‡‡‡‡‡‡	242	7
Test execution time (seconds)	Pre: 158 Load: 223 Post: 68	282
CPU utilization	3% to 10%	~75%

6.3.2 Load generation capacity

The BAT load application resulted in a lower average load compared to the S6a load application as seen in Table 13. This is reflected in the poor CPU utilization of the traffic generators by the BAT load application (see Figure 63 and Figure 64), and the generated CPS that varied from a *minimum of 0 to a maximum of 385* in the duration of the test (see Figure 68) with an *average* of approximately *200 CPS*. However, it should be noted that the BAT load application is designed to be executed as a *traffic mix* as discussed earlier in this chapter. Its load generation capacity during the test could have been significantly *reduced* during the test as only the Initial Attach traffic was run from the traffic mix (To evaluate this difference we would need to either implement more traffic cases in our load application or learn further details of the BAT load application).

The S6a load application on the other hand showed a great deal of *scalability* in load generation. Initially, it was configured to run with one LGen and executed using only one traffic generator which generated an average load on the HSS of *9%*. Subsequently, the execution was distributed over two traffic generators (using two LGens) which generated an average load on the HSS of *17.525%* as shown in Table 13. The generated CPS also scaled up from *55* (in case of one LGen) to *110* (in case of 2 LGens). Moreover, Figure 56 and Figure 57 show a very *steady CPS* during the test execution which has resulted in a *better average load* on the TSP cluster. Thus, in the future the load generation capacity can be

‡‡‡‡‡‡‡ The total number of operating system processes during the test execution

easily *scaled up* by increasing the number of traffic generators. It should also be noted that the S6a load application generated an average load on the HSS of around *49%* when it was deployed and tested using a high capacity traffic generator (refer to section 5.5.3.2). This clearly proves the potential of the S6a load application to generate even higher loads, provided that a high capacity traffic generator is used and a traffic mix (with more traffic cases in addition to the Initial Attach are implemented) to sustain the load generation for longer periods.

Table 13: CS/BAT versus DS/S6a load generation capacity

Parameter	CS/BAT result	DS/S6a result
Total number of LGens used	200	2
Average load on the (HSS) TSP DICOS TPs (%)	5.576	17.275
Maximum CPS	385 (for 200 LGens)	63 (for LGen1, 59 for LGen2)
Minimum CPS	60 (for 200 LGens)	53 (for LGen1, 53 for LGen2)
Average CPS	~200 (for 200 LGens)	~110 (for 2 LGens)

7 Conclusion

The results of this thesis project clearly demonstrate the benefits of the distributed scheduling architecture of the TITANSim framework. For the S6a Initial Attach traffic case, the S6a load application produced an average load of 17.275% on the HSS compared to the BAT load application which produced an average load of 5.576% on the HSS. The physical memory consumption of the S6a load application was less than one third of the physical memory consumption of the BAT load application. The S6a load application designed and tested in this project shows high scalability in load generation and high efficiency in terms of utilization of test system resources. It is possible to extract even higher performance from the S6a load application by adding features such as Load Regulation and Remote Transport which will be discussed in chapter 8.

A decision to adapt the distributed scheduling architecture for system test of the HSS at Ericsson will be the management's prerogative. If such a decision is made, then there will be a need for migration of the existing test modules for the HSS (which are currently using the central scheduling architecture) to the distributed scheduling architecture. This is a highly complex task in itself considering the required framework updates to TITANSim and the sheer number of FSMs that will have to be designed in order to carry out the migration.

The S6a load application can be used as a model, or extended in the future, when the test modules for the HSS are migrated from the central scheduling architecture to the distributed scheduling architecture. The migration project would consist of creating new application libraries in TITASim, extending some of the existing application libraries of TITANSim, and creating the FSMs (traffic cases) for the test modules. After the migration to the distributed scheduling architecture, creating new traffic cases (using FSMs) for the system test of the HSS would become simpler and faster as there will be no TTCN-3 coding/compilation. An exciting possibility of the future would be a TTCN-3 test case generator tool that can create TTCN-3 test cases these FSMs. Such a tool will also significantly reduce the time spent on test framework development (refer to section 8.4).

8 Future work

The TITANSim framework provides many features that were not utilized in developing the S6a load application due to the thesis project's limited scope and time constraints. In this chapter, we will discuss some important features that the S6a load application should have in future when it is used in a production environment within Ericsson.

8.1 Load regulation

The TITANSim CLL provides a base load regulator component, which should be extended by the S6a load application and initialized with a reference to functions that measures the SUT load. This would make it possible to regulate the target CPS based on the *current load level* of the SUT (i.e., the CPU usage of the SUT). The next CPS target is calculated based on the *previous CPS* value and the *last two load levels*, according to the specified load level in the configuration. A schematic view of the load regulation is shown in Figure 69. The CPS value is regulated only if the SUT load level reaches or exceeds the target load level set in the configuration. Below that value the original CPS is used **without** any regulation.

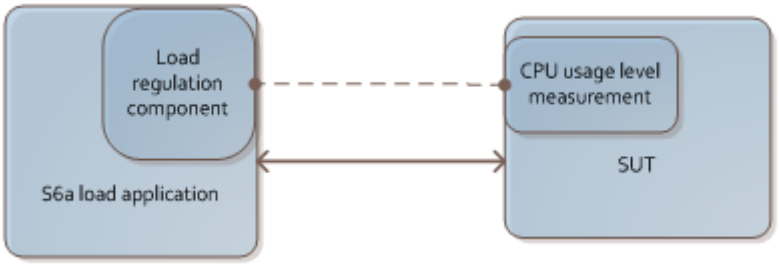


Figure 69: Load regulation for S6a load application

8.2 Remote transport

As discussed in section 5.4.4, the DiameterS6a_LGen component extends the Diameter_LocalTransport component for communication with the transport layer. By extending the local transport component the DiameterS6a_LGen component itself performs the transport functions, such as set up and tear down of transport connections towards the Diameter Proxy. This can be considered as an *overhead* affecting the *LGen performance* since the DiameterS6a_LGen is primarily responsible for generating only the S6a protocol traffic. In the future it is recommended to use the Diameter_RemoteTransport component, hence the DiameterS6a_LGen component will be relieved of performing the transport functions. The remote transport component uses a Diameter_Mapper component that performs the transport functions on *behalf* of the DiameterS6a_LGen component as shown in Figure 70.

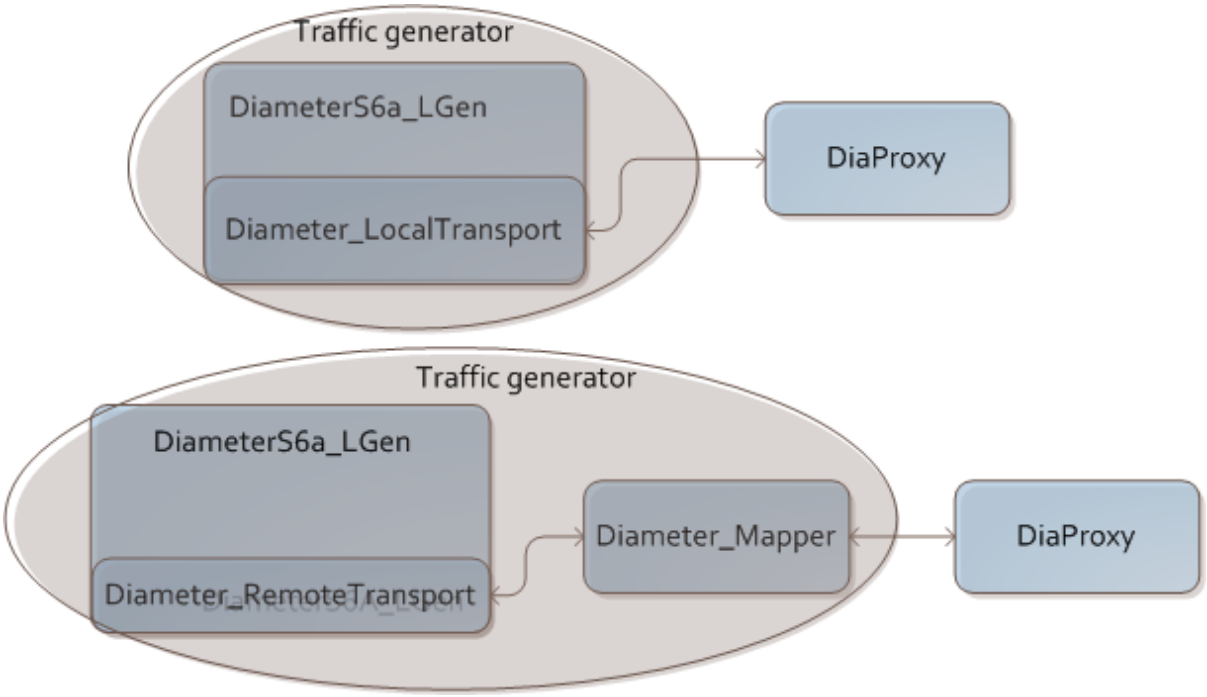


Figure 70: Remote transport mechanism for S6a load application

8.3 Application extension

The existing BAT load application for the ESM module contains a number of traffic cases for Initial Attach, Detach, Handover LTE to 3G, ESM user profile management, etc. For this thesis project the scope of the project was restricted to the Initial Attach procedure. Hence, the S6a load application in the future should be extended to support the other procedures as well. This will require updates to the Diameter_Types module, Diameter application library, and a number of FSMs will need to be created for each of the traffic cases.

When other traffic cases are created in future, it will be possible to execute a scenario with multiple traffic cases. Each traffic case can also be configured to carry a certain weight in the target CPS (for example, traffic case 1 shall have 10% weight, traffic case 2 shall have 30% weight, and traffic cases 3 shall have 60% weight in the target CPS).

In the future the S6a load application should also be configured to run the traffic in three phases namely, pre-execution phase, load phase, and post-execution phase. During the *pre-execution* phase transport connections are established and other configuration updates to SUT are performed to prepare it for the load phase. The desired load of protocol traffic is generated during the *load phase*. In the *post execution* phase clean up actions (for example, tearing down of transport connections, freeing used entities, test verdict calculation, etc.) are performed to conclude the analysis. This partitioning will facilitate evaluation of the system while it is under the desired load and explicitly excluding the warm up and shutdown of the testing process itself.

8.4 TTCN-3 test case generator

Patrick Wacht, Thomas Eichelmann, et al. have proposed a test development process in the publication titled '*A New Approach to Design Graphically Functional Tests for Communication Services*[§§§§§§§]' [34]. In this test development process, a behavior model is created (by the test developer) using the FSMs. The FSMs are created based on the service description (use-cases). The test case generator tool uses the behavior model to create new TTCN-3 templates and also utilizes pre-defined TTCN-3 templates to generate the test cases. The authors have also proposed a *connectivity concept* that the test case generator tool can employ to map the data from the behavior model to the corresponding TTCN-3 element. Apart from generating the test cases, such a tool also generates TTCN-3 templates based on the service description. These generated templates can be integrated into the test framework. This presents an exciting opportunity for automation of test framework development.

[§§§§§§§] The publication can be directly accessed from http://www.taunusportal.de/etechnikorg/aufsaetze_vortraege/aufsaetze/wacht_et_al_comgeneration_paris11_.pdf

References

[1] TTCN-3.org. (Undated). *About TTCN-3*. [Online]. Available: http://www.ttcn-3.org/Origin.htm. Last access on 4/8/2011

[2] TTCN-3.org. (Undated*). TTCN-3 Application areas*. [Online]. Available: http://www.ttcn-3.org/ApplicationAreas.htm. Last access on 4/8/2011

[3] TTCN-3.org. (Undated). *TTCN-3@Ericsson*. [Online]. Available: http://www.ttcn3.org/TTCN3UC2007/Presentations/Wed/1500_ttcn_users_at_ericsson_a.pdf. Last access on 4/8/2011

[4] TTCN-3.org. (Undated). *TTCN-3 open source tools*. Available: http://www.ttcn-3.org/OpenSourceTools.htm. Last access on 4/8/2011

[5] János Zoltán Szabó and Tibor Csöndes. *TITAN TTCN-3 Test Execution Environment*. Test Competence Centre, Ericsson Hungary Ltd. January, 2007 [Online]. Available: http://www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf. Last access on 4/8/2011

[6] Gábor Ziegler. *Runtime test configurations for load testing*. Lecture notes. Ericsson Hungary Ltd. 15 May, 2007 [Online]. Available: http://www.ttcn3.org/TTCN3UC2007/Presentations/Fri/Session%204/ziegler-Run-time_test_configurations.pdf. Last access on 4/8/2011

[7] Gonzalo Camarillo and Miguel-Angel Garcia-Martin. *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and Cellular Worlds*. John Wiley & Sons, 2004, 354 pages, ISBN 0470 87156 3.

[8] ETSI Standard. *TTCN-3 Core language standard*. ETSI ES 201 873-1 V4.2.1. July 2010

[9] Colin Wilcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3.* John Wiley & Sons, 2005,

282 pages, ISBN-13 978-0-470-01224-6.

[10] ETSI Standard. *TTCN-3 Standard Part 5: TTCN-3 Runtime Interface*. ETSI ES 201 873-5 V4.2.1. July 2010

[11] ETSI Standard. *TTCN-3 Standard Part 6: TTCN-3 Control Interface*. ETSI ES 201 873-6 V4.2.1. July 2010 3GPP. *IP Multimedia (IM) Subsystem Cx and Dx interfaces; Signalling flows and message contents Rel. 10*. 3GPP TS 29.228 V10.0.0. December 2010

[12] 3GPP. *IP Multimedia (IM) Subsystem Cx and Dx interfaces based on Diameter protocol; Protocol details Rel.9.* 3GPP TS 29.229 V9.3.0. September 2010

[13] 3GPP. *IP Multimedia (IM) Subsystem Mobility Management Entity (MME) and Serving GPRS Support Node (SGSN) related interfaces based on Diameter protocol Rel. 10.* 3GPP TS 29.272 V10.1.0. December 2010

[14] P. Calhoun, J. Loughney, E. Guttman, and G. Zorn and J. Arkko. *Diameter Base Protocol.* Internet Request for Comments. 2070-1721. RFC 3588. September 2003 [Online]. Available: http://www.rfc-editor.org/rfc/rfc3588.txt. Last access on 5/15/2011

[15] Szabados, Kristóf. *Structural Analysis of Large TTCN-3 Projects*. Testing of Software and Communication Systems. Lecture Notes in Computer Science. Springer, Berlin, 2009, pages 241-246. Available: http://dx.doi.org/10.1007/978-3-642-05031-2_19. Last access on 5/15/2011

[16] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. *SIP: Session Initiation Protocol.* Internet Request for Comments. 2070-1721. RFC 2543. March 1999 [Online]. Available: http://www.rfc-editor.org/rfc/rfc2543.txt. Last access on 5/15/2011

[17] Zhang Li. *Service Improvements for a VOIP Provider*. Master of Science Thesis. TRITA-ICT-EX-2009:104, August 2009 [Online]. Available: http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/090829-Zhang-Li-with-cover.pdf. Last access on 5/15/2011

[18] Victor Ferraro-Esparza, Michael Gudmandsen, and Kristofer Olsson. *Ericsson Telecom Server Platform.* Ericsson Review No. 3. 2002 [Online]. Available: http://www.ericsson.com/ericsson/corpinfo/publications/review/2002_03/files/2002032.pdf. Last access on 5/31/2011

[19] M. Kucherawy. *Message Header Field for Indicating Message Authentication Status*. 2070-1721. RFC 5431. April 2009 [Online]. Available: http://www.rfc-editor.org/rfc/rfc5451.txt. Last access on 5/31/2011

[20] Dan Peterström. *IP Multimedia for Municipalities: The supporting architecture*. Masters Thesis, School of Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm, Sweden. TRITA-ICT-EX-2009:103, August 2009 [Online]. Available: http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/090818-Dan_Peterstrom-with-cover.pdf. Last access on 5/31/2011

[21] 3GPP. *IP Multimedia (IM) Subsystem Mobility Management Entity (MME) and Serving GPRS Support Node (SGSN) related interfaces based on Diameter protocol Rel. 10.* 3GPP TS 29.272 V9.5.0. December 2010

[22] George Din, Sorin Tolea, and Ina Schieferdecker. *Distributed Load Tests with TTCN-3.* Lecture Notes in Computer Science. Volume 3864/2006, 2006, pages 177-196, DOI 10.1007/11754008_12 [Online]. Available: http://www.springerlink.com/content/5240k24061512350 Last access on 7/15/2011

[23] Ina Schieferdecker, George Din and Dimitrios Apostolidis. *Distributed Functional and Load Tests for Web Services.* International Journal on Software Tools for Technology Transfer (STTT). Springer-Verlag, Berlin, Heidelberg, ISSN 1433-2779, Volume 7 Issue 4, August 2005, pages 351-260, DOI 10.1007/s10009-004-0165-6 [Online]. Available: http://www.springerlink.com/content/tkyuwjg8ned0wb42/ Last access on 7/15/2011

[24] Markus Warken. *From Testing to Anti-product Development.* International Journal on Software Tools for Technology Transfer (STTT). Springer-Verlag, Berlin, Heidelberg, ISSN 1433-2779, Volume 10, Number 4, July 2008, pages 297-307, DOI 10.1007/s10009-008-0074-1 [Online]. Available: http://www.springerlink.com/content/745567223522m2px/ Last access on 7/15/2011

[25] Robert K. Wysocki. Effective. *Software Project Management.* Wiley Publishing, 2006, ISBN-13: 978-0-7645-9636-0.

[26] Ken Schwaber, Mike Beedle. *Agile Software Development with Scrum.* Microsoft Press, 2004, ISBN: 0-7356-1993-X.

[27] Ferenc Bozóki, Tibor Csöndes. *Scheduling in Performance Test Environment.* 16th International Conference on Software, Telecommunications and

Computer Networks (SoftCOM 2008), September 2008, pages 404-408, DOI 10.1109/SOFTCOM.2008.4669519 [Online]. Available: http://ieeexplore.ieee.org/search/freesrchabstract.jsp?tp=&arnumber=4669519&queryText%3Dload+test+ttcn-3%26openedRefinements%3D*%26filter%3DAND%28NOT%284283010803%29%29%26searchField%3DSearch+All. Last access on 7/15/2011

[28] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Professional, 1997, ISBN-13: 978-0-201-63392-4.

[29] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling Software with Finite State Machines.* Auerbach Publications, 2006, 362 pages, ISBN-13: 978-0-8493-8086-0.

[30] Peter Brass. *Advanced Data Structures.* Cambridge University Press, 2008, ISBN: 978-0-521-88037-0

[31] Wikipedia. (July, 2011). *Modulo Operation.* [Online]. Available: http://en.wikipedia.org/wiki/Modulo_operation. Last access on 7/16/2011

[32] International Telecommunication. Union (ITU). (Undated). *Introduction to ASN.1.* [Online]. Available: http://www.itu.int/ITU-T/asn1/introduction/index.htm. Last access on 7/16/2011

[33] International Telecommunication Union. (Undated). *OSI Networking and System Aspects – ASN.1.* [Online]. Available: http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf. Last access on 7/16/2011

[34] Patrick Wacht, Thomas Eichelmann, Armin Lehmann, and Ulrich Trick. *A New Approach to Design Graphically Functional Test for Communication Services.* 4th IFIP International Conference on New Technology, Mobility and Security (NTMS), February 2011, pages 1-4, DOI: 10.1109/NTMS.2011.5721068 [Online]. Available: http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5720565%2F5720575%2F05721068.pdf%3Farnumber%3D5721068&authDecision=-203. Last access on 8/12/2011

TRITA-ICT-EX-2011:185

www.kth.se