BUILDING A GENERIC STRESS TOOL FOR SERVER PERFORMANCE AND BEHAVIOR ANALYSIS

Student: Ada Lho-la Casanovas León

Supervisor: Jordi Torres Viñals

Co-supervisor: Javier Alonso López

Department: Computer Architecture

Field: Telematic Networks and Operating Systems

This project has been realized at the Conrad Zuse Zentrum in Berlin in collaboration with Professor Artur Andrzejak during a CoreGrid Research Exchange Program.

Barcelona School of Informatics,

Technical University of Catalonia (UPC),

September 2008.

Acknowledgments.

I would like to begin expressing my gratitude to the people who made my participation in a research exchange program possible.

I wish to thank Jordi Torres, my supervisor, for the vote of confidence and taking special care of my situation, of me. I also am very thankful to Javier Alonso for his patience and support, and for taking the time to pick up the phone and call me whenever it was necessary. Many thanks to Professor Artur Andrzejak for the leap of faith of taking me in after a very brief encounter, for his patience and kindness, Dzieki :). Professor Dr. Alexander Reinefeld for welcoming me to his house and having an excellent taste for chocolate. I warmly acknowledge the CoreGrid board, specially **Céline Bitoune** for trusting me and watching my back.

The following thanks are addressed to the people who made Berlin my second home.

Kathrin Peter, for laughing at my jokes and stopping all her important tasks to let me in her office and listen with a quiet smile. Stephan Plantikov, for an inexhaustible optimism, and an impressive knowledge of Java. **Daniel Mauter**, my spanish speaking Oasis at the ZIB! **Mikael Höggvist**, always ready to help, and in case of failure, always ready to make me laugh. All of you, admirable brains with an admirable person inside, danke, tack.

Last but not least, I wish to thank the people to whom I really owe the success of this project.

To my sister **Laura**, the most beautiful thing I've ever had. To my mother, **Pepa**, for always reminding me I'm the most beautiful thing she's ever had. To my second mother **Rosa**, for giving me the chance of having something that is unique, doubled. To my father for a warm hand, to **Josep Casanovas** for an honest one, both are priceless. To **Alba Coll** for something so big, words would spoil it. To **Abel Salgado** for always being there and listen to my rabble, and **Carles Mata** for not doing so, thank you really!

Finally I thank **Viktor Lant**, my faithful mirror, for grounding my feet and making my mind fly. Szerelek minden.

Table of Contents

Section 1

Introduction

This section discusses the context and motivation of this thesis, along with its scope and main objectives.

1 Overview

What is this about?

The past decade has witnessed a significant evolution in the use of the Internet. What once was a revolutionary mean of obtaining and sharing information has now become a rather common environment where services thrive. Services through the Internet span from a simple customer/provider paradigm to multi-tiered provider chains that constitute symbiotic business (and research) communities. In the early years after the Internet became mainstream, organizations were focused on actually being *Online* to expand their presence or offerings. The unaccustomed and fascinated user was by then tolerant and mostly learning how to navigate through this new opening of business possibilities. Performance or fault issues were managed adding more hardware or delegating more responsibilities to system administrators. Nowadays these factors can no longer be neglected; in order to stay competitive, service providers must keep an eye on the quality of their services, or customers will run away to competition.

Why are performance and quality so important?

It is within the very nature of the Internet to nest an exponential flow of information. Communities proliferate and fade constantly (forums, blogs, news sheets...) where experts and professionals share their insight on a wide array of topics and subjects[1]. In the Internet community, feedback and reputation of a service is likely to spread without necessarily an actual intervention of the provider itself. Benchmarks[2] display a clear and thorough study on how services behave, using several metrics to compare and evaluate their quality. They appear as key components of SLA (Service Level Agreements) which are contracts between customers and providers where the quality of the service (QoS) is expressed in terms of specific values for availability, throughput capacity, latency and other metrics. Corporations like SPEC (Standard Performance Evaluation Corporation) or TPC (Transaction Processing Performance Council) have focused for years on the specification and design of standardized benchmarks. Additionally, several tools have been designed with the same purpose: to put a given system under a magnifying glass and point out its flaws by emulating hundreds of users.

What is the problem?

Whereas there is a solid theoretical background in the establishment of benchmark specifications, an analytical model that schedules requests to a server will not represent a real scenario. Modelling is, nonetheless, an effective and fast way to setup request distributions. Benchmarks use different types of test models :

- The Stress Test aims to determine the limits of a system by using a constant burst load or increasing it until the server crashes.
- The “Real Load” Test uses an estimated load to test the system, focusing on the study of performance metrics under normal load to ensure the established QoS is met. HP's LoadRunner for example, uses a Controller to capture real request flow and reproduces it using load generators.
- Other methods are generally used, such as increasing the load step by step, using mathematical distributions, or altering it depending on feedback performance data from the server.

With the exception of products like LoadRunner that is able to reproduce real data obtained

[1] This phenomenon is also known as the Network effect, where the more users there are of a service, the more value/quality this service gains to new users.

[2] A benchmark is a quantifiable measure of the performance of a computer or computer-related product, used for comparison with the performance of similar products.

from a server and nevertheless requires recording it first, most benchmark implementations are based upon artificially generated load using approximations to reality. The orchestration of such tests is easy, JMeter for instance, allows users to create several blocks defining a constant load and to specify when the emulation should jump from one block to the other, or whether it should use a specific distribution (e.g. negative exponential).

As many others, these applications are of great use to test the performance of a service, nevertheless when it comes to generating real load, e.g. reproducing a server output, they provide limited support, or none at all.

What are the contributions?

The purpose of this master thesis is to design and implement a tool which is able to perform a stress test of a service under real load. In more detail, taking as input the traces of a real load or some concise description of a synthetic load, our tester generates a series of requests and issues these requests via a pool of distributed, emulated clients. Our design allows for a flexible configuration of the type of test as well as the type of service. Additionally it can be extended and tuned to fit the specific purposes of the user.

2 Origins and Motivation

What was the original problem?

Prior to this project we needed to perform a load testing experiment against an Axis server. Axis is an implementation of SOAP (Simple Object Access Protocol), in short, it is a framework for generating and deploying Web Service applications. The scenario of the experiment had to be built on real data obtained from benchmarks. More specifically, we were handed data on the number of requests a server was receiving and servicing at a given time. The rates for these requests changed every minute so it was impossible to use a software that could not read the server log and change the rates when needed. Several performance testers were overviewed and considered too complex to use.

The original motivation for this project appeared while working on a research collaboration between the UPC and the ZIB computing center. The objectives of this research were to study service behavior to predict faults and determine when rejuvenation techniques should be applied.

Another interest of study was to test if an overloaded service with decreasing performance can recover after a certain time if the load is reduced. Results obtained from this study would help autonomic computing researchers to build adaptative self-healing solutions, that is, mechanisms by which a component can detect a malfunction and make the necessary steps to correct it and return to a normal state.

Our project was then funded by the CoreGRID Research Exchange Program (REP), CoreGRID is a Network of Excellence aiming to setup collaborations between researchers with the purpose of accelerating research in the area of Grid and Peer-to-Peer technologies. The REP has been realized at the ZIB in collaboration with Professor Artur Andrzejak, with the purpose of building an application that could emulate our scenario and using it to fulfill our research objectives.

In its early stages, the program ran on a single Java Virtual Machine (JVM), after reading the input it would take the number of requests and sparse them in time according to the real data, then it would reproduce them creating a thread for each request. A thread-based system satisfied our needs for concurrency. After several experiments, and even adding several JVMs that would run in parallel, we encountered performance issues in the client machine. In our setting it was not possible to overload a machine with another machine of equal characteristics without overloading the latter[1]. The next step was then obvious, a distributed system was necessary to generate load from multiple client machines, and that is how this project came to be. What started as a single Java class has evolved to a set of more than 20 classes and interfaces allowing for a flexible, generic and distributed benchmarking tool.

3 Purpose

What do we expect from our implementation? The mechanics.

The Client-Server paradigm can be divided into what is done on the consumer side, what is done on the server side, and what is perceived by either of them. Typically the client will send messages containing instructions to the server. For each type of application or framework there is a different way of creating these messages. For example, a Web Form captures input values and sends them to the server, in the simplest of cases in plain text, on more elaborate ones building a structure to wrap the data. On the other hand, more complex applications like Java Web Services use a piece of code installed on the client machine, that will communicate with its counterpiece on the server side to fulfill the consumers' needs.

[1] Actually the client machine would overload before generating any significant load on the server.

Our stress tool has been built to accept any service client, that is, users can build a small Java program that executes the request for the service they wish to test, and anchor it to ours. With the schedule extracted from the input data we can generate requests using the anchored service client, emulating a real scenario. Figure 1.1 depicts the basic architecture of our tool. As mentioned earlier, in order to generate a significant amount of load, our goal is to distribute the request generation among several machines as shown on Figure 1.2.

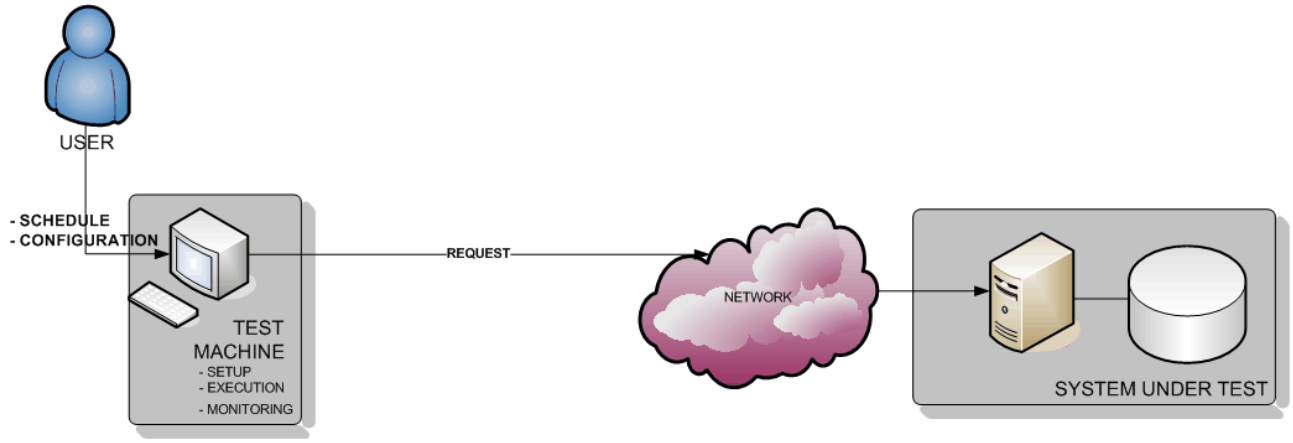


Figure 1.1 : Basic Architecture and Interactions

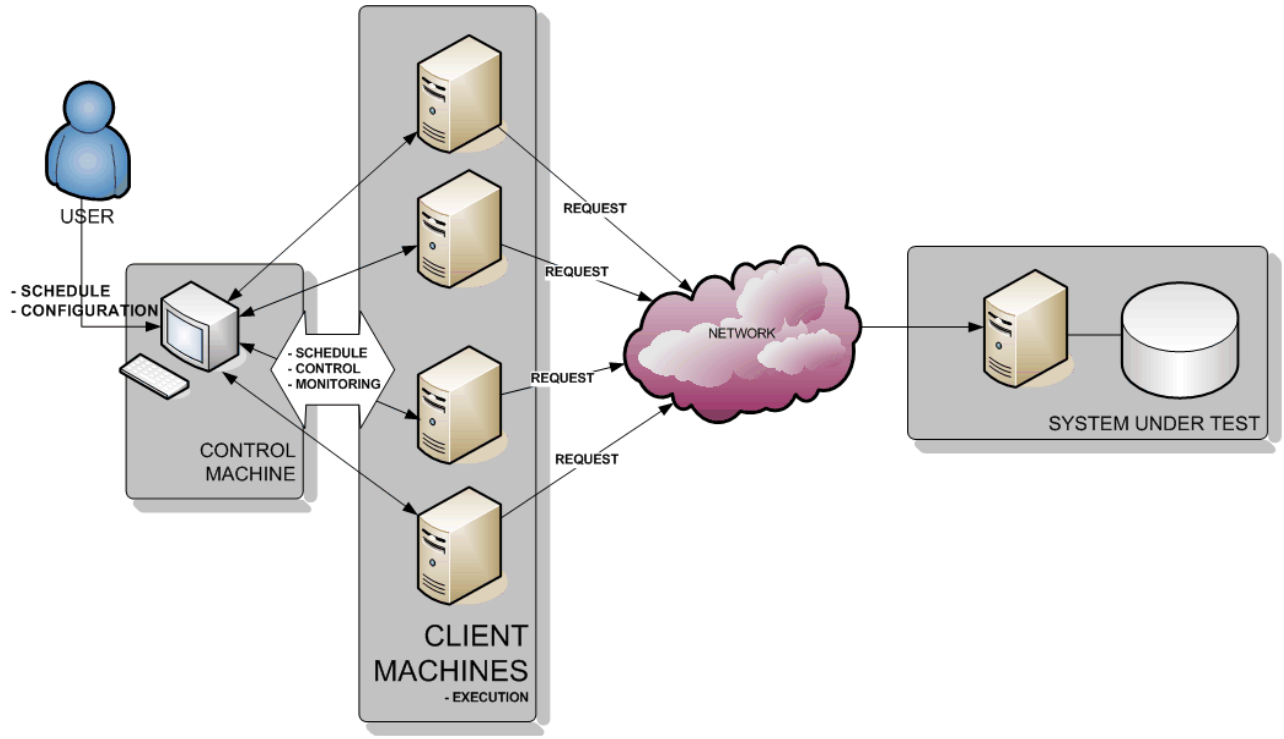


Figure 1.2 : Distributed Architecture and Interactions

One of the key elements of the tool is the management and scheduling of the requests distributed among multiple clients. Each client machine has its own schedule derived from the original data in the preprocessing step, and will execute requests in accordance to it. In order to manage this schema one of the machines is granted the responsibility of scheduling and distributing the tasks, as well as supervising their execution. The distribution and communication process is transparent, resulting in an execution that could have been done on a single machine.

Further sections of this document describe the particular techniques used to achieve our objectives. In the following paragraphs we discuss the additional aspects that we found interesting to keep in mind.

User-Oriented Thinking

It has been one of our main concerns throughout this project to be able to offer a solution that would be easy to understand and use. Our target users must be familiarized with the service they wish to test, as well as have a basic knowledge of Java, but they won't have to know how each part of the program works, nor write their own scripts if they do not wish to. We have wanted to keep it simple, using a mechanism to tune the program by changing fields on an xml file.

Community-Oriented Thinking

As far as our experience goes, any software built for a purpose (or even for general purpose) does not always match the expectations of all users. Therefore we have oriented the design towards keeping ours extensible, and furthermore, replaceable. We provide users with interfaces to each class, allowing advanced users to insert their personal implementations replacing or enhancing the existing ones. Moreover, it would be gratifying to see that the community extends, adjusts and improves it, which would mean it is useful, and being used.

4 Walkthrough

In order to make the document "reader-friendly", from this point on the tool will be called Tester and our standard user will be Alice. Alice is working at a high performance computing center that offers services to external stakeholders. The center is about to add a new service to the cluster and Alice is responsible for tunning its performance. She will first assess its limits by evaluating its behavior under burst distribution. After this first phase, the evaluation has to be narrowed to a

realistic load, Alice has access to logs and statistics on the current load of other services and is interested on testing hers with the same patterns.

Throughout the following Sections we will describe the evolution of the project and how it will help Alice. Section 2 browses through the requirements and objectives we established, while commenting on the choices we made. In Section 3 we will describe several similar tools and their possibilities, as well as their interesting features and drawbacks. Sections 4 and 5 concern the process of building the Tester. Finally the calendar and cost evaluation of the project will be found in Section 6 along with conclusions and other comments.

Section 2

Initial Study

Alice visits us exposing her problem and the features she needs for her experiments. This Section establishes the foundations of our project by outlining its characteristics, as well as the requirements we will take in consideration.

1 Overview

The Tester was not initially conceived to run on a Distributed System, nor to accept any kind of input other than the csv values it did in its first version. Initial requirements have been progressively modified, new requirements have appeared. This Section is dedicated to those decisions, in the lines below we will expose the problem and, using the Alice example, the requirements she would need the Tester to fulfill.

1.1 Concepts

We now present a definition of several concepts that have an important role in the design of the Tester. The descriptions presented here underline what aspects of these concepts we consider, broader definitions can be found in other contexts.

Benchmarking:

We refer as benchmark to the scheduled execution of an application using different scenarios and configurations with the purpose of collecting metrics on its performance and availability.

Benchmarking tool:

A Benchmarking tool is a software that targets application designers or administrators as end users, that is, it requires a basic knowledge of the system that is being put to test. It provides with the functionalities required to perform and configure a benchmark. Generally a benchmarking tool offers a "plug and play" mode for simple tests as well as the means for users to extend its functionalities.

Distributed System:

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a
single, integrated computing facility.

Cluster :

A cluster is a distributed system, a collection of computers -generally of equal characteristics- connected through local area network for minimal overhead. The purpose of a cluster is to execute ressource exhausting jobs by distributing them among several computers and processors. The distribution can be seemless, as if the job had been executed at a single computer, therefore in most cases computers in a cluster share a file system.
Our software has been tested at the ZIB D-Grid cluster :

Dell-Cluster: 5 Teraflop/s peak, 960 Gigabyte Main Memory, Infiniband A Linux-Cluster composed of 80 compute nodes with 480 cores. 40 nodes are connected by Infiniband with a latency of about 4 µs and a bandwith of 1.2 GByte/s.

JVM:

The Java Virtual Machine is a software portable to any platform that supports the execution of java bytecode. Users can program in Java or other languages and compile their code to java bytecode, then run it on one or several JVMs on the same computer. Java provides with libraries to monitor JVMs ressource consumption.

PBS scheduler :

The D-Grid Cluster uses Torque as the Batch Queuing System. A PBS (Portable Batch System) is a software for scheduling jobs among distributed computational resources. Jobs are generally executed by submitting a PBS script to the batch queue. This script contains information on the amount of nodes and processors to allocate, the duration of the execution, and other job behavior attributes.

Server:

A server is a component running on a machine that offers services to any computer that can access to it. In the course of this project we have worked both with Apache Axis and Jakarta Tomcat. The Axis server is used to deploy and execute WebServices whereas Tomcat is a servlet container associated with an HTTP server that runs on a JVM.

Service:

In the context of this master thesis a service consists of any action performed at a remote server that provides data or business logic upon request from clients. In short, a service is any information, software or system hosted by a server that clients can access by sending requests to this server.

A Web Service is a "software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards"[1]. In short, it is an application that provides an interface to allow other applications to interact using XML messages and the Web.

Servlets are hosted at the Tomcat Container, they represent a remote java program that can be invoked through an HTTP request using GET or POST methods. Their container is responsible for deploying and mapping them to a URI.

---

[1] Definition from the WWW consortium at http://www.w3.org/TR/ws-arch/

1.2 Objectives

Our user Alice needs to *benchmark* different types of *services* hosted on a *server* in a *cluster*. Since she is testing the machine there is no specification on the type of service she wishes to use, therefore we can experiment with different possibilities. In order to test the server, she can use as many machines in the cluster as she needs which allows for a *distributed* benchmark. The cluster is managed by a *PBS scheduler* that can schedule jobs in parallel, it is then not necessary to manually start the Tester on one or several *JVMs* in all nodes, using a PBS script we can request to execute it upon resource lease.

The constraint in this feature is that the Tester must have a single execution module that is capable of running in all machines. If we wish to add roles to the different machines we must find a way of doing so after the execution has begun.

What characteristics does Alice want in our tool?

As mentioned earlier, Alice has collected data on the load of other servers in the cluster at the Computing Center. The data has been preprocessed and now consists of a csv (comma separated value) file containing thousands of lines.
For each line there are two values:

- A *timestamp* which corresponding to the system time in miliseconds of the server machine when the data was logged.
- The number of requests being processed at the given timestamp.

We are requested to design a tool that will take these two values as input, and reproduce the load. We will also provide with means of modifying the load, either by changing the rates or by adding information to the input derived from the csv file. Additionally we will explore the different possibilities to accept any type of input other than the csv *timestap-number of requests* pair to allow for a more versatile benchmark setup.

1.3 Decisions

We now present the initial choices we had to make before begining the project design. Even though design is independent from implementation, it has proven useful to us to have an idea of the *how* to know exactly the boundaries of what we can do.

Why use Java?

We have chosen Java as the language for our implementation. There are several discussions on whether to use Java or not in research environments, and the subject on "Why Pick Java?" has populated the community and is still in debate.
Since we do not intend (and are not qualified) to make a methodic exposition of the pros and cons of Java, we will present our particular reasons:

1. Java is easy: Java offers an extended documentation through its APIs. Additionally there are several tools making Java programming simple and fast.
2. Java is popular: It is easy to find information on the Internet concerning java. It is broadly used in several domains and by different types of developers, which makes solving particular issues much easier.
3. Java allows for extensions of its classes in a fashion that helps developers to use and reuse Java code without having to read a single line of the actual implementation.
4. It is easy to translate a standard UML model to Java classes, and vice-versa.
5. The JVM allows us not to worry about making specific implementations for each operating system, or even hardware related.
6. There are several monitoring tools for the JVM, which is interesting in further extensions of the Tester, where Clients can be monitored and brought down or up depending on their performance.

All these features make up for the traditional argument that points at Java for having an additional overhead, at least in the scope of this project. Moreover, by using exclusively Java to program the Tester, most of these features are inherited, as we have stated earlier it is our main purpose to make a simple and easy tool. Java is nowadays the default (and generally first) language learnt in computer science environments, so it is not very probable to find developers who are unable to understand it. We do not intend to discredit other languages, we have chosen Java for the comfort it provides us, it would be good news to see the Tester ported to other languages in the future.

Why use XML as input?

The Extensive Markup Language (XML) is a very practical method to label and organise information in a hierarchical fashion. It is commonly used to build messages in aplications, store data in XML databases, create configuration files, Ant build files... Additionally it is human friendly, which is why so far it represents our user interface. It replaces a command-line input and allows for the implementation of a graphical interface that will read from and write to the xml configuration file.

RMI vs CORBA.

The Java Remote Method Invocation (RMI) mechanism and the Common Object Request Broker Architecture (CORBA) are the two most important and widely used distributed object systems. Both specifications work using a stub object at the client that communicates with a skeleton on the remote object.

The main advantage of CORBA is that it allows a system to work with objects written on many different languages using the Interface Definition Language (IDL). The IDL defines the methods by which client and remote object will communicate, independently of the language they are programmed. This is particularly important for interfacing to existing systems
which could be written in any language.

On the other hand Java RMI is much simpler, being 100% java it had the advantage to support features like creating local copies of remote objects, perform distributed garbage collection, Exceptions, java interfaces etc.

Given the Tester is a java software, we didn't have the need to support other languages. Java RMI allows us to quickly setup and modify remote services, its simplicity is interesting to us since we want to allow users to modify and extend our RMI classes if they need to.

2 Requirements

In the following lines we will present both the functional and non-functional requirements for the Tester. As a standard software development procedure, these requirements are categorized and numbered for easy reference.

2.1 Functional Requirements.

The functional requirements present the features our software must provide with to guarantee our design satisfies Alice's specifications.

Requirements **[FR-001]** to **[FR-005]**: Main Functionalities

[FR-001] The Tester must be capable of reading an input csv file containing timestamp values and the corresponding requests being processed by a server at that given time.
[FR-002] The captured values will be pre-processed and stored as a schedule of requests.
[FR-003] The Tester must supply with the means to change the scale of the captured values, to allow for a "speedup" or "slowdown" of the emulation.
[FR-004] The schedule must be then followed and reproduced, by means of requests against a given server.
[FR-005] Information on the latency and throughput will be collected to provide data on the server's behavior.

Requirements **[FR-006]** to **[FR-010]**: Distributed Features

[FR-006] The Tester must be capable of running on several machines, and create means of communication between them.
[FR-007] The Tester will provide with means of discovery or allow for manual input of the location of the given machines.
[FR-008] Supervision and setup of the distributed instances of the Tester will be done from a lead machine, users are given the possibility of choosing it.
[FR-009] The schedule will be generated at the leading machine and distributed in a round-robin fashion amongst several client machines.
[FR-010] All machines must be synchronized with the central one, to allow for a coherent time

reference.

Requirements **[FR-011]** to **[FR-012]**: User Input

[FR-011] All user input must be read from an XML file prior to any GUI implementation.
[FR-012] Users must be able to customize the topography of the system, and tune the performance of the software by means of several available variables in the XML file.
[FR-013] By filling a field of the XML file, users must be able to specify the source of data (csv file), as well as the service they are willing to test.

Requirements **[FR-014]** to **[FR-015]**: Developper Resources

[FR-014] The software must provide with interfaces to all its classes to allow for custom implementations.
[FR-015] The choice between one implementation or another must be available by changing a field of the XML file.

Most of these requirements are illustrated in Figure 2.1, the image depicts the order in which they were added to the project. The box marked as "Any input" corresponds to an additional requirement under development which allows for any type of input format. The Tester is capable of reading any type of format and storing it in a data structure, allowing developers to use it in their custom implementations.

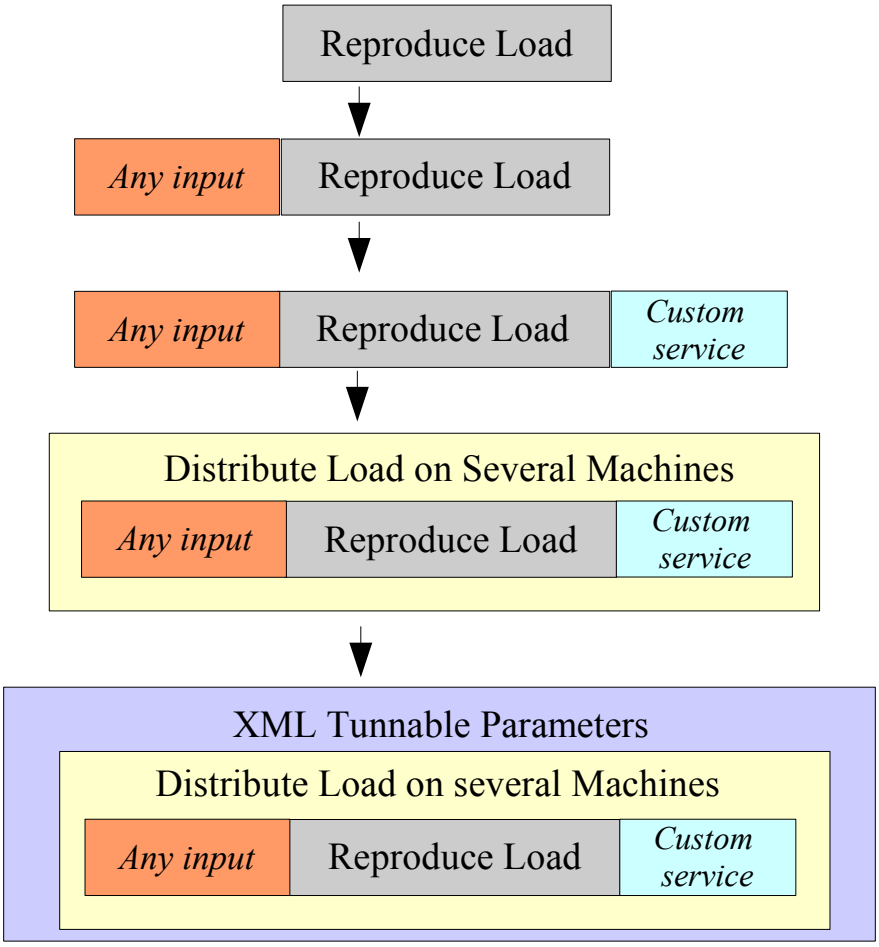


Figure 2.1 : Requirement Evolution

2.2 Non-Functional Requirements.

Requirements **[NFR-001]** to **[NFR-004]**: User Interface and Human Factors

[NFR-001] The Tester assumes the standard user is familiarized with using and programming in java, this software targets server administrators or researchers

[NFR-002] An advanced user, called Developer, is also taken into account for extensions and improvement of the software.

[NFR-003] The Tester must be easy to learn, and simple to understand.

[NFR-004] The Tester must provide with an XML input file, for user input. Users can specify an output file for results and error messages.

Requirements **[NFR-005]** to **[NFR-006]**: Documentation

[NFR-005] A user Manual must be provided.
[NFR-006] Additional documentation, such as Java APIs and Extension reports must be standarized to comply with a community prospective.

Requirements **[NFR-007]** to **[NFR-008]**: Hardware Considerations

[NFR-007] The Tester must be portable to any platform.
[NFR-008] Client machines can be regular 34bit Pcs, but the program is designed to run in Clusters and other research environments.

Requirement **[NFR-009]**:Performance Characteristics

[NFR-009] The Tester must be capable of emulating the requests at their scheduled time without overloading the Client Machines.

Requirements **[NFR-010]** to **[NFR-011]**: Error Handling and Extreme Conditions

[NFR-010] The Tester will be designed assuming its users understand its functionment, therefore are able to react to exceptions and errors.
[NFR-011] Error handling will be implemented in a future GUI.

Requirements **[NFR-012**] to [**NFR-013]**: System Interfacing

[NFR-012] The Tester must provide with interfaces to all its functional classes.
[NFR-013] Interfaces to the Client Machines must be capable of operating independently as services.

Requirement **[NFR-014]**: System Modifications

[NFR-014] The Tester will be designed to accept add-ons, modifications, and plug-ins to other benchmarking software.

Requirement **[NFR-015]**: Security Issues

[NFR-015] All machines running the program must be allowed to communicate with one another, or at least allow the leading machine to communicate with them.

Requirements **[NFR-016]** to **[NFR-017]**: Resources and Management Issues

[NFR-016] Installation instructions must be provided.
[NFR-017] Test and Debugging classes must be provided.

Section 3

Related Work

The Tester was built out of need for a simple solution to our problem, Several existing tools that perform stress tests and benchmarks were overviewed to determine if they could provide with the functionalities we required. In this section we will briefly expose their main features and the reasons why we discarded using them.

1 Foreword

The Tester project was not initially a master thesis, it was the quick solution to our need for a benchmarking tool that would satisfy Alice's requirements. The applications we present in this sections are medium scale projects, some of them initiated by large software organizations, others backed up by powerful companies, and all developed by a team of experts. In the following lines we will describe how they work in general, focusing on the aspects that we found interesting.

2 HTTPerf

What is it?

Httperf is a tool for UNIX-Like Operating Systems to measure web server performance.

How to setup a test?

Httperf gathers scenario configuration data through argument input in a Unix terminal. Users can specify several configuration aspects:

- A server address or a file containing a list of URIs to reproduce.
- Load distribution parameters such as the rates or whether to ramp up, the duration of the test, number of calls or connections etc.
- Test case parameters such as information about sessions.
- Other configuration parameters to increase the test performance by modifying how httperf interacts with the operating system.

Execution

BASIC EXAMPLES

```
httperf --hog --server=www
```

Creates one connection to host www and requests the root document (http://www/)

```
httperf --hog --server=www --num-conns=100 --rate=10 --timeout=5
```

As above, except that 100 connections are created at a fixed rate of 10 per second

```
httperf --hog --server=www --wsess=10,5,2 --rate=1 --timeout=5
```

Generates a total of 10 **sessions** at a rate of 1 **session** per second. Each session consists of 5 calls that are spaced out by 2 seconds.

```
httperf --hog --server=www --wsess=10,5,2 --rate=1 --timeout=5
--ssl
```

As above, except that httperfcontacts server www via SSL at port 443 (the default port for SSL connections).

```
httperf --hog --server=www --wsess=10,5,2 --rate=1 --timeout=5
--ssl --ssl-ciphers=EXP-RC4-MD5:EXP-RC2-CBC-MD5 --ssl-no-reuse
--http-version=1.0
```

As above, except that httperfwill inform the server which cipher suites to use (EXP-RC4-MD5 or EXP-RC2-CBC-MD5); also, httperf will use HTTP version 1.0 which requires a new TCP connection for each request; additionally, SSL session ids are not reused, so the entire SSL connection establishment process occurs for each connection.

Figure 3.3 : Basic Examples of httperf usage

httperf reports performance metrics for all experiments with different types of output:

- Total :
    - Number of TCP connections initiated
    - Number of requests
    - Number of replies
    - Overall time spent testing

- Connection
    - Connection initiation rate/period and the max number of concurrent connections
    - Lifetime statistics for successful connections
    - Average time to establish a successful connection
    - Average number of replies received on each connection that received at least one reply

- Request
    - Shows the rate and period at which HTTP requests were issued.
    - Average size of the HTTP requests in bytes

- Reply
    - The minimum, mean, max, standard deviation of the reply rate and number of samples used in calculation
    - The average time for the server to respond to a request and the average time it took to fully receive the reply
    - Average length of reply headers, content, and footers
    - A histogram of received status codes

- CPU : Summarizes the CPU statistics over the course of the experiment.

- Network I/O used to transfer and receive all data throughout the experiment.

- Errors

- Session
    - Average number of connections per session (usually 1 with persistent connections)
    - Average time to complete session successfully
    - Average time before an unsuccessful session has failed
    - Number of sessions with 0, 1, 2, etc replies during the session

Thumbs up!

Quick setup of stress tests

Low overhead

Exploits Unix System features to increase performance

Rich output metrics

Thumbs down!

Httperf cannot handle more than approximately 1020 concurrent connections, since it is the limit imposed by the maximum amount of file descriptors in Linux

Platform dependent

Can't schedule requests

Hard to extend

3 JMeter

What is it?

This application, developed by the Apache Jakarta project team, is a 100% java tool initially designed to test Web Applications. Latest versions can test the performance and functional behavior of many different server types, java applications, protocols, static resources etc.

How to setup a test?

A Jmeter scenario is represented by a *Test Plan* which contains all the components necessary to perform a test. Users can build scenarios by adding and configuring components to the Test Plan's hierarchy :

The first element of a Test Plan is the *Thread Group*, it controls the threads that will execute requests against a service. Users can configure parameters like the *number of threads*, the *ramp up period* (time between the execution of the first thread and the time when all threads are running), and the *amount of times* the test should be executed. New versions add parameters like *start time* and *end time* to limit the execution duration.

A thread group typically contains *samplers* to specify the type and content of a request and *logical controllers* to change the order of requests, determine which sampler to use or modify the content of a request.

Listeners are used to collect the result data of the experiments, they can be configured depending on the metrics users wish to obtain.

Jmeter sends requests without delays, users can add *Timers* to specify the delay between requests, these distributions can be exponential, gaussian, constant, etc.

One of Jmeter's peculiarities is the possibility for the user to develop his/her own controllers, listeners and samplers in order to add them to a Jmeter test plan. The new sampler is added simply in Jmeter by adding the JAR file in a specific directory.

Distributed Execution

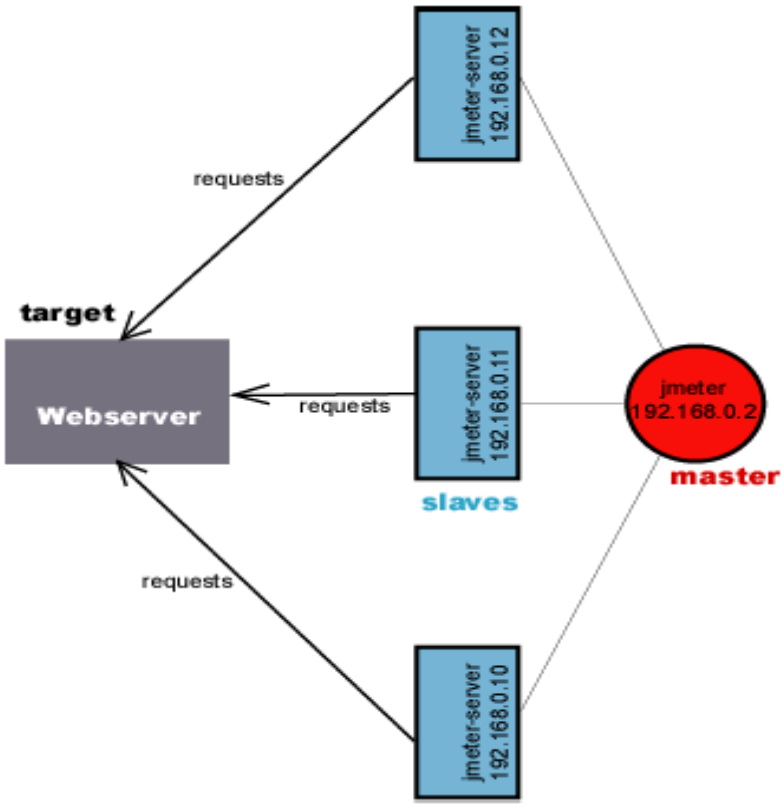


Figure 3.2 : JMeter Distributed

A test run may be distributed on several computers which have an RMI registry running. However it is only possible to start remote computers one by one, and stop them one by one too. Using listeners, it is possible to record results in a graph which will record the average time of the response, the throughput, etc. and also export them to a file.

Thumbs up!

Distributed testing

Platform Independent

Java Extensibility: Pluggable components allow multiple testing possibilities

Thumbs down!

Extending Jmeter is not a complicated task but requires time, the architecture and terminology are often inconclusive. After several hours of study we couldn't determine if it was possible to extend the Timer to collect the input data from a file and distribute it among several remote machines in different thread groups.

It is necessary to start a specific executable on every machine one by one.
Remote machines must be declared in a property file before starting the application.

4 LoadRunner

What is it?

Mercury Interactive's LoadRunner is a load testing tool that analyses system behavior and performance. It exercises the entire enterprise infrastructure by emulating thousands of users and employs performance monitors to identify and isolate problems. LoadRunner is distributed by Mercury Interactive company under a commercial license.

The target systems are various, and most existing technologies are supported, like Web servers, Web application servers, streaming media servers, databases servers, Java, ERP, ...
Another Mercury Interactive product is available for evaluation. AstraLoadTest is a part of LoadRunner, but it is designed only to check web servers scalability. The AstraLoadTest software proposes three different modules specialized in creating a scenario, running a scenario and viewing the results.

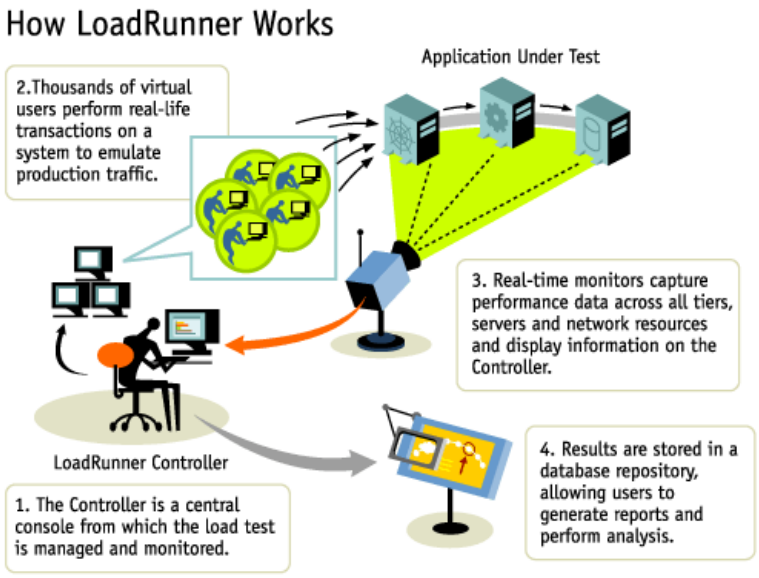


Figure 3.3 : How LoadRunner Works

How to setup a test?

Building a scenario is quite simple with *VUGen* (Virtual User Generator) which generates C-language script code to be executed at the virtual users by capturing network traffic between Internet application clients and servers . The result is the creation of a schedule structured as a tree, with each action to execute. It is also possible to manually add interaction elements to build a hierarchical test plan.

There are two main types of scheduling, but both allow to customize a ramp up period, a duration period and a ramp down period. The schedule by scenario or schedule by group are two different visions for running the test.

- The schedule by scenario allows for instance to start a given number of virtual users every fixed time.
- The schedule by group allows to delay a test on a remote host, by setting the start time after another host started.

Another functionality is a *rendezvous* point which determines when all the virtual users simulated have reached this point (or the time out period).

Execution

The Controller manages load test runs based on *scenarios* invoking compiled VUGen scripts and associated settings. During runs, the status of each machine is monitored by the Controller.

Once the test is finished, results are collected from all the distributed computers, and merged into one document. This report summarizes some key results such as number of users emulated, total throughput, average throughput, total hits, average hits, response codes and transactions information. These data are also available on charts. A lot of graphics are available depending on the targeted system.

Monitoring provides several system values such as the percentage of processor time, number of threads, memory usage but it is only available for NT or Linux systems.

Basically, the test results displayed represent the response time for each action, nevertheless it is possible to merge actions in a transaction and monitor results comprised between the transaction *start* and *end* tags.

Thumbs up!

VUGen allows to capture scenarios and compile them into scripts.

The Controller monitors all machines involved, including the server.

LoadRunner has been adapted to several systems.

Thumbs down!

Real scenarios must be captured before reproducing them and not extracted from raw input. It is possible to build Virtual User configurations and script to specify a scenario but they have to be built for each Virtual User manually.

The schedule of requests can't be extracted from an input file and distributed in a custom manner among Virtual Users without parsing, distributing and creating configuration files manually.

LoadRunner is not extendable.

Loadrunner is distributed under a commercial license

5 The Grinder

What is it?

The Grinder is a 100% java load testing framework that simulates client requests to an application. It consists of three types of processes:

- **Agent processes:** A single agent process runs on each test-client machine and is responsible for managing the worker processes on that machine.
- **Worker processes:** Created by the agent processes, they are responsible for performing the tests.
- **The console:** Coordinates the other processes and collates statistics.

How to setup a test?

Scenarios are written in Jython (Jython is a Python interpreter allowing manipulation of Java objects) therefore inheriting Java's flexibility and extensibility. User Scripts can be created by recording actions of a real user using the TCP Proxy. The script can then be customised by hand. Simple scenarios can be composed into more complex scenarios by allocating a percentage of the total requests to a specific application use case or different workloads for specific times of a day.

Input data (e.g. URL parameters, form fields) can be dynamically generated. The source of the data can be anything including flat files, random generation, a database, or previously captured output.

On each agent, users must copy a property file that contains all information for the test (eg. console address, number of threads, etc.) and start the Grinder agent. On one host users must start the Grinder console to control other agents.

Execution

The console broadcasts to the agents on the network the instruction to start the test. Each agent creates a log file with the result of each transaction (HTTP code…), it can also save the result page in HTML. Another file contains time response of each transaction with the thread number, the number of runs, the response time and if the transaction is successful. The result file can be used with a spreadsheet like Excel in order to construct graphs or calculate other statistics.

Grinder only monitors transactions every second. Each agent sends information to the console during the load test.

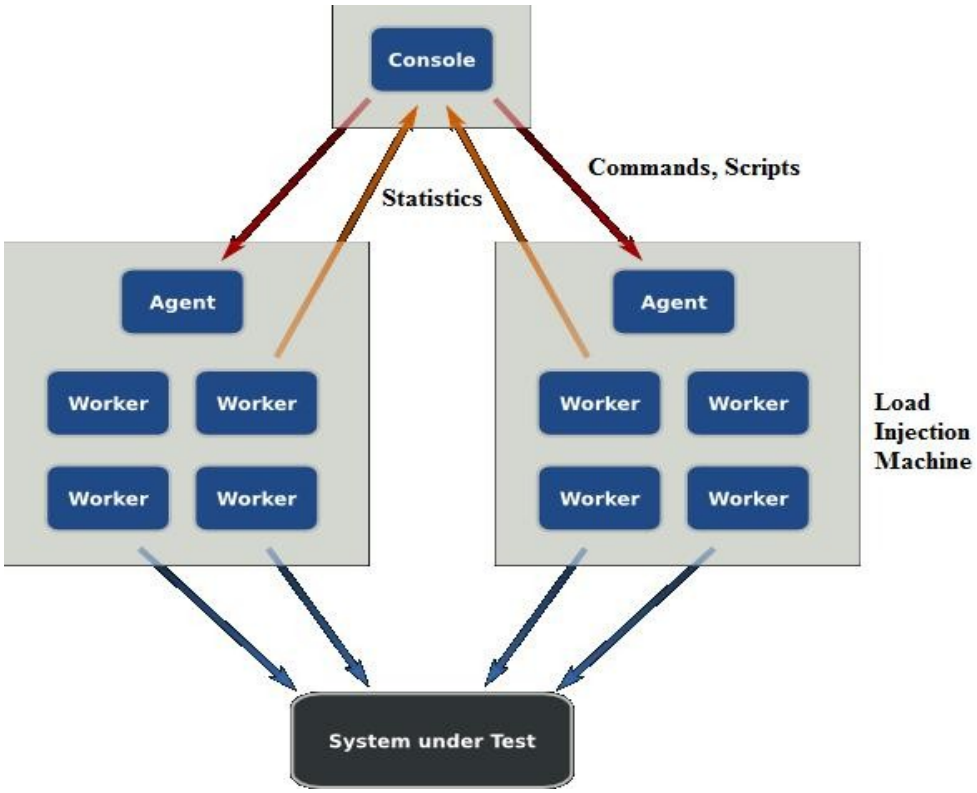


Figure 3.4 : The Grinder Architecture

Thumbs up!

Jython : easy dynamic generic design of scenarios + Java's object oriented possibilities.

Thumbs down!

No scheduling of requests.

6 PushToTest

What is it?

TestMaker is a framework and utility that builds intelligent test agents which implement users interactions in several environments. This program is developed in Java and is available as open source. TestMaker turns unit tests into functional tests, load and scalability tests, and service monitors automatically. It allows several levels of customisation (librairies, XML files, and more...) and provides with output data processing utilities.

How to setup a test?

Unit tests (test agents) can be programmed directly in Jython or generated using a Firefox add-on or a network proxy that will record all actions. It is possible to program an agent to use threads for parallel execution, or sequentially, as well as continuously loop, create a thread per user etc.

The largest of TestMaker's libraries is TOOL (Test Object Oriented Library), and it includes classes for handling several communication protocols: HTTP, HTTPS, SOAP, POP3, JDBC, etc. to allow for a specific implementation of unit tests.

Execution

TestMaker can be executed from the command-line, test agents are then executed by an automation system. In addition, TestMaker bundles the Apache Axis TCPMonitor tool, which allows monitoring of HTTP exchanges on a specified port.

Users can trace operations in a specific window using a Jython command and advanced users can handle their own result representation in a Java frame with graphics, using the appropriate Java libraries.

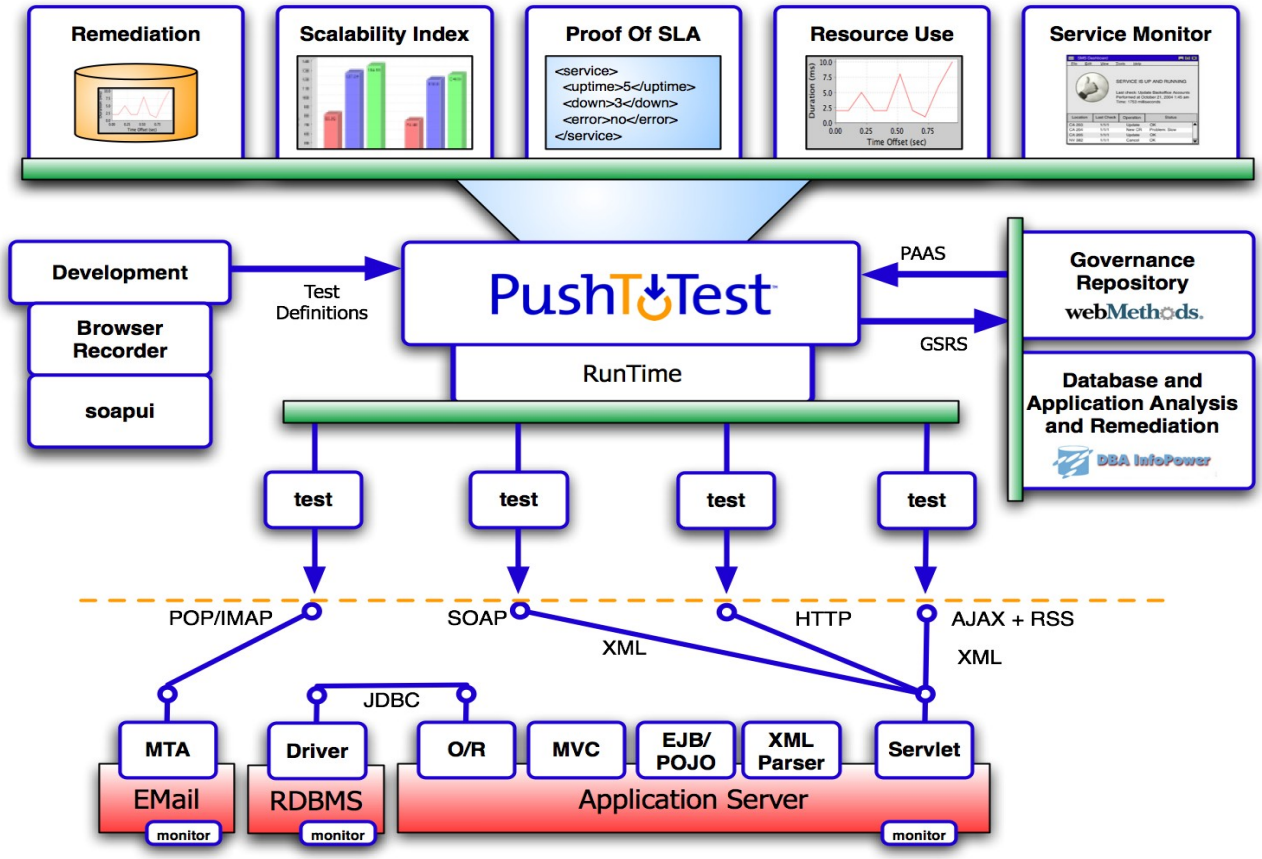


Figure 3.5 : PushToTest Architecture

Thumbs up!

Jython : easy dynamic generic design of scenarios + Java's object oriented possibilities.

Can record network traffic and browser interactions and generate test agents.

Agent Wizard that creates skeletal agents with the structure of a test agent that can be completed to create a scenario.

Thumbs down!

Test agents must be programmed using Jython in order to perform any test.

Requests can't be scheduled.

Very complex to adapt to our purposes.

Distributed feature is limited by a commercial license.

7 Clif

What is it?

CLIF is a Java framework providing a generic infrastructure to generate load on any kind of system, and gather performance measurements (request response times, computing resources usage). Its goal is to overcome a number of typical limitations of existing similar projects, especially in terms of versatility. CLIF's infrastructure should:

- be independent from the system under test and its associated invocation protocols;
- not enforce any specific definition model of load injection scenarios;
- be able to generate high loads using an efficient distributed injection;
- be suitable and adaptable for a great range of user skills and needs (plain users, advanced users, developers)
- be user friendly and support centralized deployment, control and monitoring features of distributed injectors and probes;
- run on common operating systems supporting a standard Java runtime.

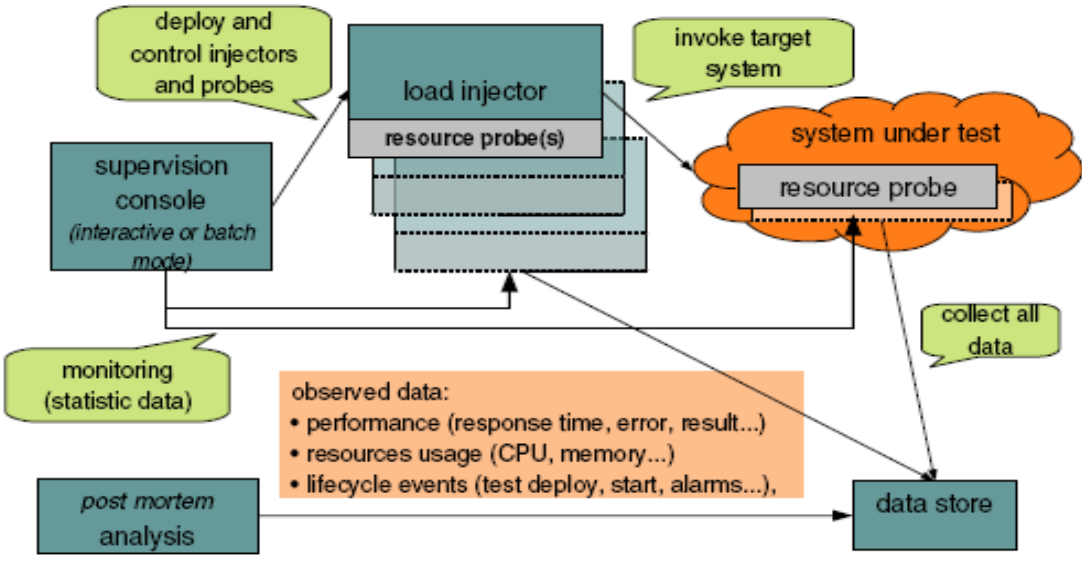


Figure 3.6 : CLIF Distributed Architecture

How to setup a test?

CLIF has been designed using the Fractal component model which allows to design, implement, deploy and reconfigure systems and applications, thus reducing the development, deployment and maintenance costs.

CLIF is composed of a single supervision console component, for test management and an arbitrary number of *CLIF server* components. These server components are empty, minimal Fractal components, in which the console may create a *load injector component* and/or a *resource probe component*.

ISAC is a scenario architecture for CLIF providing a formal support to describe elementary behaviors (e.g. typical behaviors of virtual users). Those behaviors are basically sequences of requests and delays (think times), including conditional loops, branches and preemption. Those scenarios may be written in XML, or generated either from a GUI (see figure 2) or by capturing real sessions through a proxy mechanism. A scenario combines a definition of those behaviors with a load profile specifying the population (i.e. number of instances) of each behavior as a function of time. ISAC uses a plug-in pattern to resolve its genericness and actually invoke the system under test using the appropriate protocols, as well as to implement specific conditions, specific timers, and external data sources.

Execution

The load injector is in charge of hosting and executing test scenarios and measuring response times, while the probe component measures the consumption of system resources (related to CPU, memory, swap, network, disk, etc.).

Other components cover different functionalities:

The *storage component*, which is responsible for storing test data.

The *collector component* responsible for transporting test data to the storage component.

The *analyzer component* supporting exploitation of test data and a number of *scenario components*, one per load injector, responsible for invoking the tested system.

Execution control, monitoring, and setup of load injectors and resource probes, is done from a supervision console. Supervision is achieved either through graphical user interfaces, or using command-line tools enabling batch programming of tests.

CLIF received the Lutece d'Or 2007 award for the best open source project made by a big company.

Thumbs up!

Everything

Thumbs down!

Poor documentation for developers willing to extract a schedule of requests from a file and distribute it among several injectors.

Comments: The *discovery* of CLIF was made once this master thesis was at its *point of no return*. Hadn't it been the case, we would have considered writing a plugin for CLIF instead of developing our own tool. Nonetheless we do not know if the result of using such a large framework for our purposes would have satisfied our needs or accelerated the process.

8 TPCW Java Implementation.

What is it?

TPC Benchmark™ W is a transactional web benchmark based on workload against an e-bookstore service. It uses Emulated Browsers to send multiple requests for the same session, requesting different URLs including searches, purchases etc. thus reproducing the load a real user would generate. We have adapted two ClientWrappers to work with TPCW, in the following lines we will describe the basic architecture of the TPCW implementation we have used as well as its interactions with our system.

The TPCW implementation we have studied comes with 14 servlets that allow for a variety of interactions with the server. The actions available include searching books, adding to shopcart, purchasing, login, etc. All data is stored in and retrieved from a remote database, the initial contents of the database are randomly generated. The sequence of requests is also random, nevertheless it follows a Markov chain, that is, the probability of a browser following a link depends on the page it is currently viewing.

How to setup a test?

A TPCW Java Implementation test is executed by calling the RBE with several configuration parameters as arguments, we will now list some of them :

- EB [EB factory name][number of EB]
  [factory name]=rbe.EBTPCW1Factory for Browsing Mix rbe
  =rbe.EBTPCW2Factory for Shopping Mix rbe
  =rbe.EBTPCW3Factory for Ordering Mix rbe
- OUT [filename] specify a .m output file
- RU [ramp up time]
- MI [measurement interval time]
- RD [ramp down time]
- WWW [URL]
- ITEM [Number of items] Number of items in the database, used to generate random searches.
- CUST [Number of customers] Number of customers in the database,used to generate random CIDs.
- GETIM [Request images] True will cause RBE to request images. False suppresses image requests.

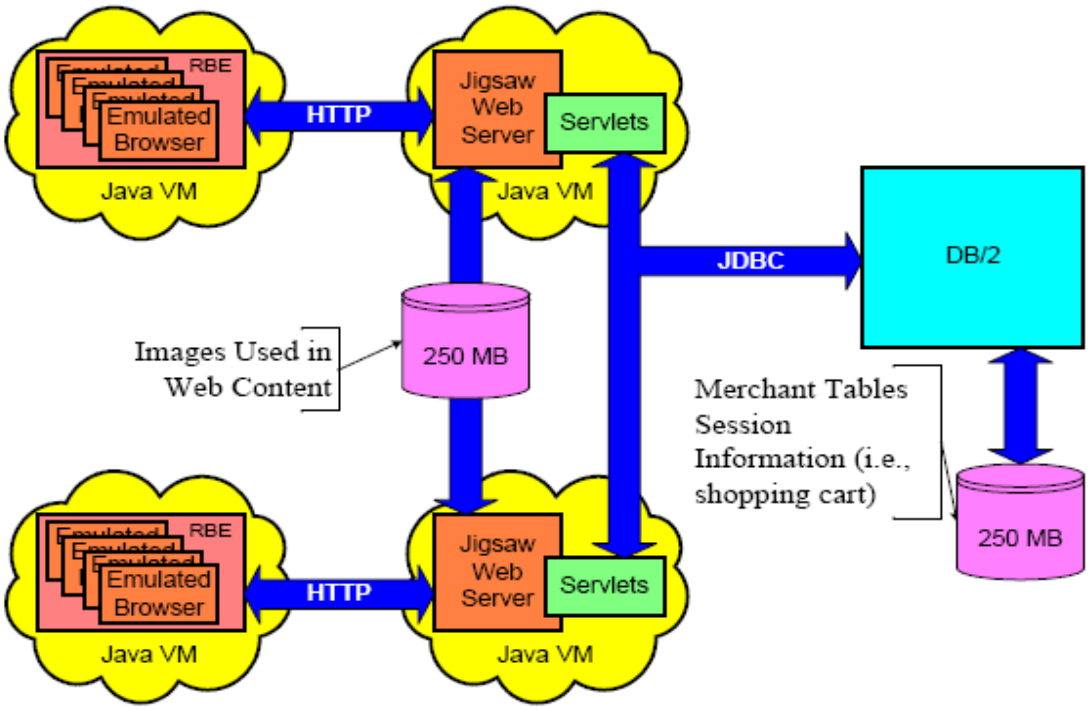


Figure 3.7 : TPCW Java Implementation at PHARM

Execution

The Remote Browser Emulator controls the execution of the benchmark, taking user configuration parameters as input, it populates a configuration information object that other TPCW components refer to. The RBE can be divided into two aspects of its functionality: on the one hand it runs, starts and controls the test, on the other it stores static data used to configure the test. The TPCW execution begins with a warm up ramp, a real test and then a cool down ramp periods. The RBE uses a Factory to create Emulated Browsers and start them, each EB can access the static information of the RBE. The TPCW Emulated Browsers run locally, they will send single session requests using markov chains to jump from one page to another and restart a new session once the current is over. The time between requests is known as thinktime, and can be calculated using a negative exponential distribution.

At the end of the execution the RBE stops all EBs and stores statistics extracted from the test in a matlab file.

Thumbs up!

Follows a specification established by TPC, offering many possibilities of e-business interaction using a transition matrix and markov chains.

Easy to run tests.

Can be distributed by running several RBE instances.

Thumbs down!

Hard to Setup the environment

Can't schedule requests

Several configuration parameters are obligatory when unnecessary

Metrics are printed and accessible only at the end of the execution

Monitoring is Platform dependent.

Bad Java programming, not extensible.

No documentation.

Comments: We have developped a plug-in for the Tester using the TPCW Java Implementation. Further information can be found on the following sections.

9 Conclusions

We have overviewed several tools designed to put a system under performance, stress or integrity test. Some of these applications are easily configurable to perform standard tests, others allow for extensions to build customized scenarios, most of them offer practical GUIs as well as visualizations of the output metrics.

Name	***Httperf***	***JMeter***	***Load Runner***	***The Grinder***	***PushToTest***	***Clif***	***TPCW***
Cross-Platform	*No(Unix)*	*Yes*	*Medium (plugins)*	*Yes*	*Yes*	*Yes*	*Yes*
License	*GPL*	*Apache*	*Commercial* *+pay for extra features*	*LGPL*	*GPL* *+ pay for extra features*	*LGPL*	*Free*
Language	Perl	Java	?	Java/ Jython	Java	Java	Java
Easy to Configure	Easy	Medium	Medium	Hard	Medium	Medium	Easy
Easy To extend	Hard	Java API	Not Extendable	Java API	Java API	Java API	Hard
Distributed	Yes[D]	Yes[D]	Yes[L]	Yes[D]	Yes[L]	Yes	Yes[D]
Real time monitoring	No	No	Yes[L]	No	Yes (custom implementations)	Yes	No
Real time load adjustment	No	No	Yes	No	No	Yes	No
Custom Load Possibilities	Poor	Medium	Rich	Rich	Medium	Rich	Medium
Extract schedule from input file and distribute it	Impossible	Impossible	Impossible	Impossible	Impossible	Very Hard	Impossible

Figure 3.8 : Summary of Related Work

[D] The tool is distributable by running multiple instances, each with their configuration but there is no possibility to distribute a global schedule.
[L] The tool's license restricts this feature.

We have summed up the main characteristics of several tools in Figure 3.8, focusing on the aspects we were interested on. Prior to the idea of designing our own Java application, we tried to modify httperf's code to fit our needs with no success. We also tried to adapt JMeter to execute a custom schedule derived from a file in a single test case, again, with negative results. LoadRunner was overviewed for its architecture but the commercial license and restrictions made it unreachable. The Grinder and PushToTest offered possibilities at first, regardless of how complex their customisation was, but were discarded for their inability to distribute the schedue without preprocessing. CLIF was found after the beginning of this project, we consider it might have been possible to adapt it to our purposes in a project of similar magnitude, nevertheless we can't assert the final result would have been positive.

Finally the TPCW java implementation offered a bundled package with a multi-tiered architecture (client emulator-servlets-database) that we found interesting and easy to maneuver. We decided to create a plugin for TPCW in our software, the idea seemed simple at first considering it was a Java implementation. We encountered several problems such as incorrect implementation and usage of an Object Oriented Language such as Java, errors in the Servlets and the database, poor extensibility, and zero documentation. Nevertheless we did make a plugin for TPCW that we will describe in the following sections. This adaptation is the first extension of the Tester which makes other plugins allowing for our software to interact with tools like JMeter or any other Java implementation conceivable.

Section 4

Architecture

This Section presents the basic architecture of the Tester describing an overview of the components, their functionalities and main interactions.

The Tester's architecture comprises four roles: BootStrap, Director, Client and Monitor. Figure 4.1 outlines their responsibilities in chronological order : the Tester will first BootStrap in all JVMs and determine their role, thus proceeding with the necessary steps to launch them. As a choice of design, both the Monitor and the Director run in the same JVM, several Client JVMs can be distributed locally or remotely. In the following paragraphs we will briefly define the basic features of each role as well as the interactions between them.

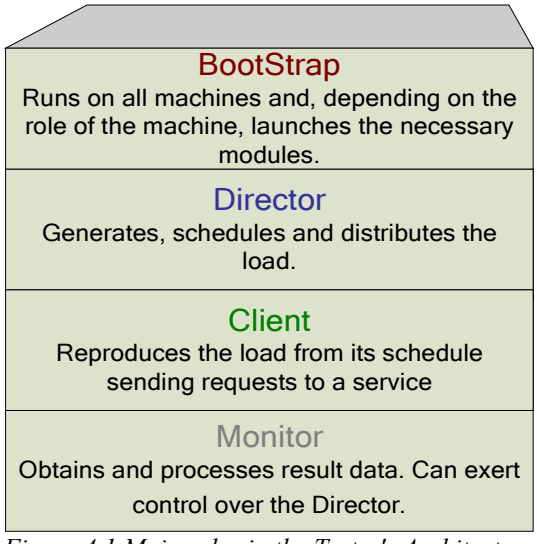


Figure 4.1 Main roles in the Tester's Architecture

1 Component Overview

The Tester has been designed to run automatically with minimal configuration and without necessarily any user intervention during runtime. Therefore, all virtual machines start the same execution whether they are Client or Director, local or remote. In order to achieve this feature, the BootStrap is responsible of locating all machines and determining which components have to be loaded. As we can see in figure 4.2, BootStrap will launch the LoadDirector, Monitor and DirectorLogger in the JVM where load generation and control is managed. Figure 4.3 presents the same process in the Client machine, where the ClientWrapper and the ClientLogger are started.

1.1 Load Generation and Distribution.

The mechanisms by which custom load input is processed and distributed are handled by three main components: the LoadDirector, the LoadGenerator and the Scheduler.

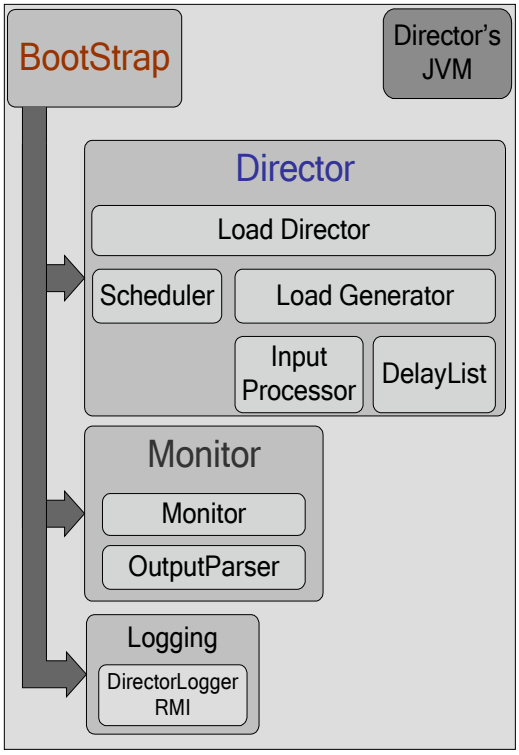


Figure4. 2: Director's JVM

The LoadDirector will locate and communicate with the clients, supervising their execution and periodically sending them schedules for the requests they have to produce against the service.

The LoadGenerator uses a *DelayList* to obtain the initial raw load (extracted from Alice's input) and an *InputProcessor* to modify or add information to the load if necessary.

Finally *the Scheduler* is responsible for a consistent distribution of the load.

1.2 Request Generation and Metric Collection.

The ClientWrapper is the remote interface to a distributed client, it will follow a schedule and send requests using the adequate ServiceClient to test Alice's server.

The ServiceClient represents one request to a service, it is a component Alice must provide to allow for a custom service test.

Output data is logged using the *ClientLogger* which dumps it to the client's local file system and can optionally forward log messages to the *DirectorLogger* that is running in the Director machine.

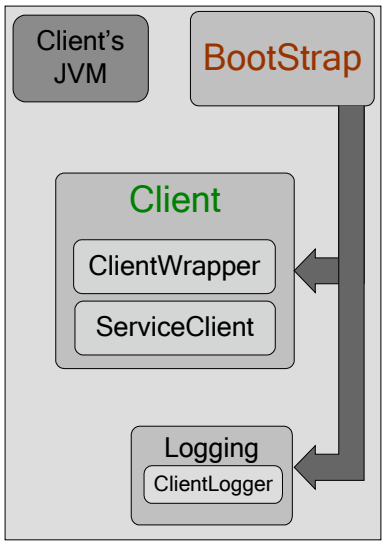


Figure4. 3: Client's JVM

1.3 Monitoring

The final step of the process is monitoring and decision making. *The monitor* runs parallel to the Director, and collects logged data to analyze it during runtime. Metrics are extracted from log messages using a standard or custom *OutputParser*. In automatic mode, the monitor can additionally decide if the configuration used to preprocess the load must be modified.

2 Interactions

We will now briefly introduce the basic actions and communication between key components of our system, adding an illustration of the flow on Figure 4.4. Once the LoadDirector is active, it establishes a link with all the clients it will control...

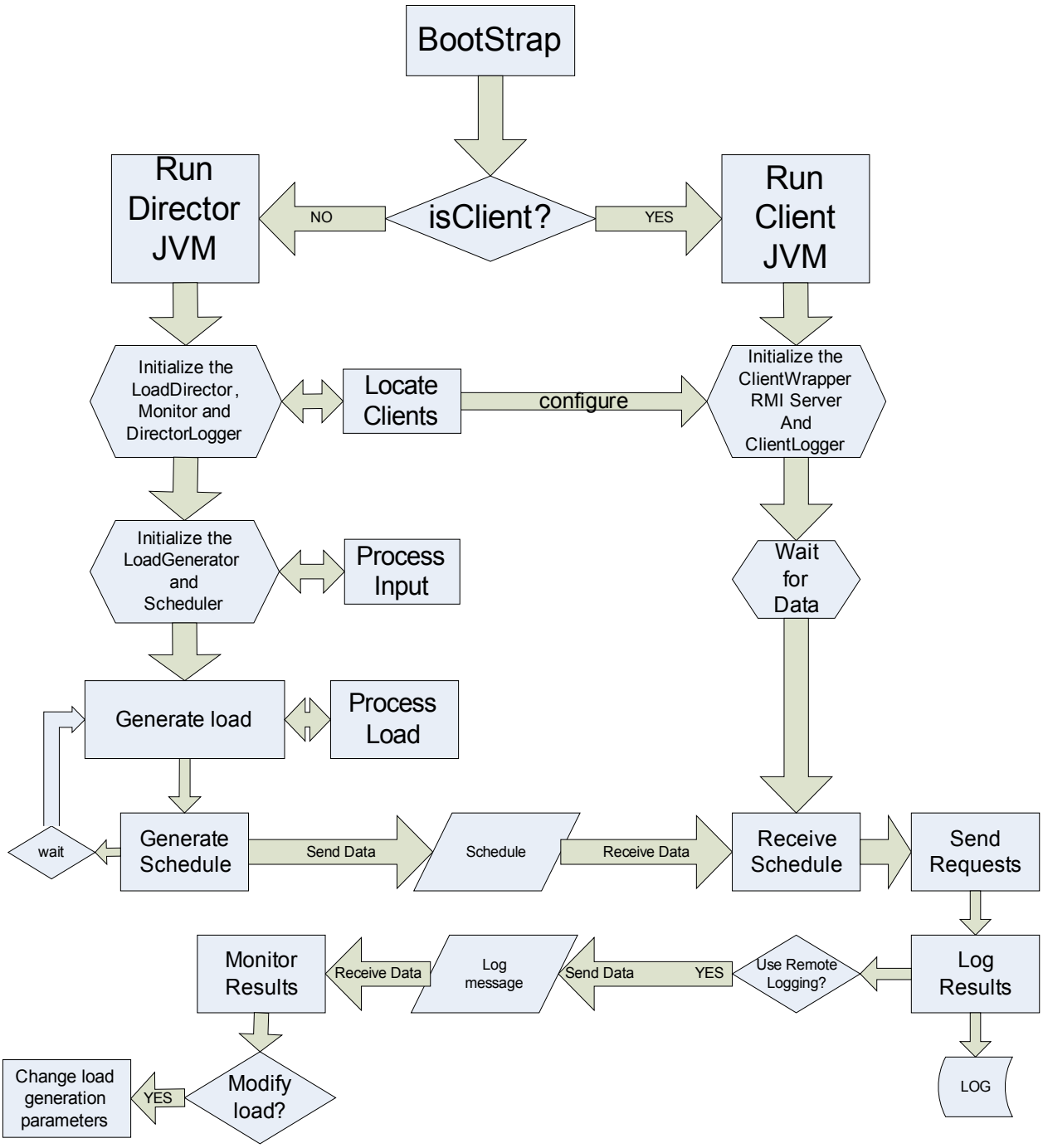


Figure 4.4 The Tester's FlowChart.

2.1 Client Lookup

After the bootstrap phase, all clients run and register a remote service then wait for commands. Meanwhile, on the Director's JVM, the LoadDirector attempts to locate and synchronize all Clients to ensure their execution is simultaneous. Once all Clients are up and ready, the LoadDirector begins initializing and configuring its load generation components.

2.2 Load Generation

Input data is collected by the DelayList whose role is to generate a list of delays, a delay represents the time elapsed between the sending of two requests. The LoadGenerator extracts parts of this list upon request and processes them with the InputProcessor, injecting additional data to the delays. The final result of this load generation is a list of tasks representing the schedule of all requests that are to be sent to the server.

2.3 Load Scheduling

Once the LoadDirector has obtained a global schedule from the LoadGenerator, it sends it to the Scheduler for it to be split into several smaller ones, one for each client. The Scheduler keeps track of the distribution order and ensures delays are adapted to be interpreted in parallel.
Additionally, whenever it is necessary to send a specific type of task to a specific client, custom distribution rules are applied at the Scheduler[1].

2.3.1 Load Distribution.

The LoadDirector sends each client a schedule of tasks representing requests using a remote method that will add them to the client's task queue. The size of the schedule should be customized according to network overhead, as well as the lapse between each sending. If a client fails to receive the schedule, the LoadDirector will try to resend and eventually drop the client.

[1] For example, requests belonging to the same session must be in the same client's schedule.

2.3.2 Load Execution.

The ClientWrapper is a remote object located in all client JVMs that is used to send requests to a service. Therefore, it allows for custom implementations to adapt it to any purpose. In our project we have designed three different ClientWrapper implementations:

The basic ClientWrapperImpl works with delays, it takes the schedule and executes the ServiceClient for each delay. The ServiceClient is a component that must be programmed by our user since it should be adapted to the particular service she is testing.

We have also designed two different ClientWrappers that combine the Tester and TPCW Benchmark. We have migrated the execution aspects of the RBE to our ClientWrapper implementations and used RBE as a static configuration component. Therefore our Clients now read user configuration parameters, create Emulated Browsers and start them following their schedule.

The Session TPCW ClientWrapper receives a schedule of tasks identified by their session ID. It then starts one EB per session and assigns the delays for that particular session. The EBs interpret these delays as thinktimes between requests. Once the session is over, the EB stops and is recycled to be used for other scheduled sessions.

The TPCW ClientWrapperImpl starts a TPCW Emulated Browser each delay and the TPCW Session ClientWrapperImpl sends requests through TPCW Emulated Browsers taking in account their session ID. These plugins for TPCW will be described more specifically later in the document.

2.4 Logging and Monitoring.

We have added basic local and remote logging utilities to our system, one ClientLogger per client and a DirectorLogger on the central machine. There are several logging methods to allow for different levels of log, nevertheless neither logger discriminate among levels. Discrimination can be done at the OutputParser, which is used by the monitor to obtain metrics from log messages. Additionally, in automatic mode the monitor can interfere with the load generation and distribution by modifying configuration parameters.

Figure 4.5 illustrates the architecture of the Tester, using a similar schema as the one presented in the introduction. In this diagram we depict only one client and one director and how they interact. (see next Chapter: Design) for more information.

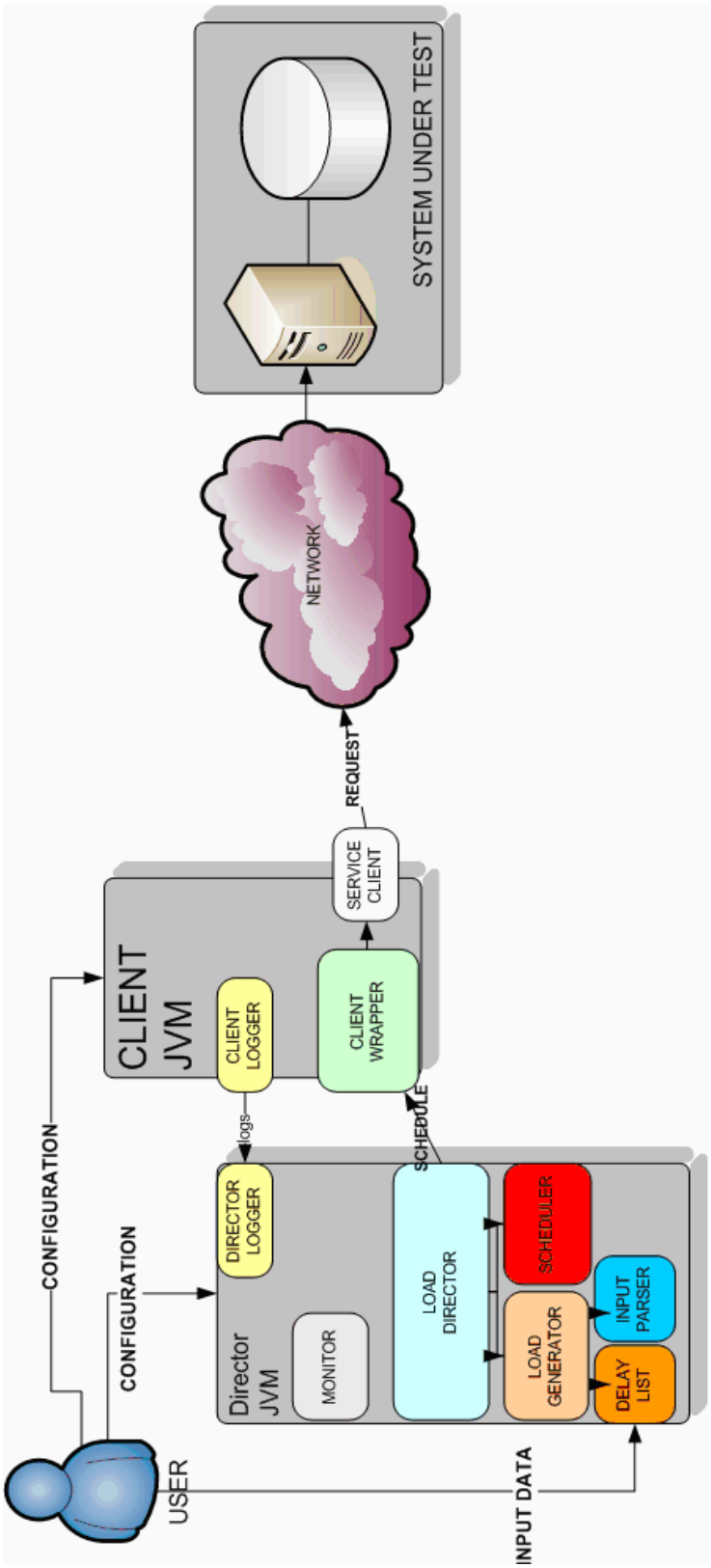


Figure 6.4 : Global Architecture of the Tester

Section 5

Design

The How and the Why: In this Section we will expose several important Use Cases for the Tester development, including an example of the possibilities it offers to advanced users. Following these descriptions, the reader will find class and sequence diagrams, an overview of the design patterns we have used as well as some comments on the choices of design and implementation.

1 Use Case Overview

As mentioned in earlier sections, once it is configured, the Tester does not necessarily require any user intervention. The benchmark begins when launching the Bootstrap and stops when there is no more load to generate. Nevertheless given the architecture, components can be started independently and used to test a server or any other application. In the following lines we will expose several use cases considering Alice takes the responsibilities of the Bootstrap. Each use case will be complemented by a small description on how the Bootstrap achieves each task. Additionally at the end of this subsection we expose several configuration possibilities that help understand the structure of the tester.

For clarity purposes, in this section we distinguish the following concepts:

Machine: one unit of execution (in this case a JVM)

Physical machine: a standalone machine that can run several JVMs, a node in the cluster for example.

Create: instantiate a component.

Initialize: configure a component to its initial state.

Initiate: activate a component. This concept is different from *run*, a component can be running yet remain inactive, waiting for its initialization.

Run: begin the execution of a component.

For quick reference, a Use Case index: (the ID value refers to the subsection where the use case is described)

ID	***Name***	***Description***
1.1.1 A	Client Discovery	The first step to start the Tester is to find all involved machines.
1.1.1 B	Configuring and starting the LoadDirector	Configuration steps of the LoadDirector component and initiation of its execution
1.1.2	Starting the Client server	The client RMI service must be started to allow for communication between the director and the client
1.1.3 A	Starting the DirectorLogger	Start an RMI service to allow for remote logging. This feature is optional, when disabled, clients only log locally and runtime monitoring is not allowed
1.1.3 B	Starting the ClientLogger	Start the logging utility in a client machine
1.1.4	Starting the Monitor	Start a monitoring utility that will supervise the execution based on the logs collected by the DirectorLogger
1.2.1 A	Generation of a set of delays	Generate a portion of requests represented by delays
1.2.1 B	Session ID injection	Create a list of LTSessionTasks based on a list of delays
1.2.2	Scheduling LTSessionTasks	Distribute of session-based load
1.2.3 A	Sending and receiving a schedule	The schedule is sent to the appropriate client and processed
1.2.3 B	Processing an LTSessionTask	Interpret and prepare the LTSessionTasks

ID	*Name*	*Description*
	schedule	received at the client
1.2.3 C	Session Execution at the EB	For each session, one EB is responsible of sending requests based on the delays received from the TPCWSEssionClientWrapper

Figure 5.1 : Index of Use Cases

1.1 Starting and Configuring the Components

The Bootstrap component offers a set of operations that allow users to start and configure components. In an automated execution Alice would only need to configure and run this utility, in a manual execution she can reproduce the Bootstrap behavior to setup the experiment using some of its methods.

1.1.1 The Load Director

Use Case: Client Discovery.

Description: The first step to start the Tester is to find all involved machines.

Preconditions: The system knows the amount of machines we are using.

Course of events:

OPTION 1: Manual input of all the machine IDs

- Alice manually inputs all IP addresses of the machines she is using.
- If she wishes to use several JVMs within the same physical machine, the IP adresses also include an additional number that we will call Port. The name was chosen because a machine's ID is represented by its IP when there is only one JVM in the physical machine, and by IP:Port when there are several, but the number itself has no functionality and is treated as an identifier, the data that must be real is the IP.
- Optionally she can specify the ID (IP or IP:Port) of the JVM she wishes to be the Director. If the Director's ID is not specified, the system chooses the machine with lowest identifier value.
- The system stores information on all machine identifiers.

OPTION 2: Client discovery using a shared file system.

- Alice uses a shared file system and sets up an IP pool folder containing one file for each client machine. The name of the file is the Identifier of the machine, composed by the IP and a number.
- Optionally she can specify the IP:Port of the JVM she wishes to be the director. If the Director's ID is not specified, the system chooses the machine with lowest identifier value.
- Alice calls the client discovery method in Bootstrap.
- The system stores information on all machine identifiers.

Postcondition: The system has information the client machine IDs and the director's ID.

Bootstrap: When Bootstrap runs in all machines and has not received information on their identifiers it takes care of generating them. Each instance writes a file in the shared ipPool folder for discovery of all clients. The only information it needs is the amount of machines involved.

Use Case: Configuring and starting the LoadDirector

Description: Configuration steps of the LoadDirector component and initiation of its execution.

Preconditions: Client Discovery phase is completed AND the current machine ID is the Director ID AND All client RMI must be running before initiating the LoadDirector.

Course of events:

OPTION 1: Manual configuration and startup.

- Alice creates a LoadDirector
- Alice creates a LoadGenerator
- Alice creates a DelayList (optional) and configures it
- Alice creates an InputProcessor (optional) and configures it
- Alice associates the LoadGenerator with its DelayList and InputProcessor.
- Alice creates a Scheduler and configures it
- Alice creates a DirectorLogger, configures and runs it.

- Alice associates the LoadDirector with its LoadGenerator and Scheduler.
- Alice initiates the LoadDirector with the list of client identifiers and the DirectorLogger.
- After the initiation the LoadDirector is active and starts running.

OPTION 2: Configuration and startup using a Preferences node.

- Alice creates a LoadDirector
- Alice creates a DirectorLogger, configures and runs it. (see subsection 1.1.3)
- Alice initiates the LoadDirector with the list of client identifiers, the DirectorLogger and a Preferences node with data extracted from an xml configuration file. Using the preferences node, the LoadDirector creates and configures all its subcomponents.
- After the initiation the LoadDirector is active and starts running.

Postcondition: The LoadDirector and all its subcomponents are configured and running.

Bootstrap: In automatic execution, the Bootstrap creates a LoadDirector and a DirectorLogger. It then initiates the LoadDirector using the list of Ids, the DirectorLogger and a Preferences node extracted from the XML file. This node contains all configuration data and will be used by the LoadDirector to create and configure subcomponents.

1.1.2 The ClientWrapper

Use Case: Starting the Client server.

Description: The client RMI service must be started to allow for communication between the director and the client.

Precondition: Client Discovery phase is completed AND the current machine ID is a Client ID.

Course of events:

- Alice creates the ClientWrapper implementation of her choice with its ID.
- Alice sets the serviceClient she wishes to use if necessary[1].
- Upon creation the ClientWrapper starts or locates an RMI registry and binds itself.
- The ClientWrapper creates and configures its subcomponents if necessary.
- The ClientWrapper runs the service.

[1] TPCW implementations do not require this step.

Postcondition: The client RMI service is up and and running waiting for an initiation signal from the LoadDirector.

Bootstrap: While running on a client machine, the Bootstrap checks several of its configuration parameters to determine which ClientWrapper implementation to start and creates it.

1.1.3 The Logging Components

Use Case: Starting the DirectorLogger

Description: Start an RMI service to allow for remote logging. This feature is optional, when disabled, clients only log locally and runtime monitoring is not allowed.

Precondition: Client Discovery phase is completed AND the current machine ID is the Director's ID AND remote logging is enabled.

Course of events:

- Alice creates a DirectorLogger implementation with the Director's ID.
- The DirectorLogger starts or locates an RMI registry and binds itself.
- Alice configures the DirectorLogger using a Preferences node **OR** manually.
- Alice starts the DirectorLogger.

Postcondition: The remote logging RMI service is up and and running waiting for logging instructions from the clients.

Bootstrap: Before starting the LoadDirector and the Monitor the bootstrap creates and configures the DirectorLogger implementation if remote logging is enabled.

Use Case: Starting the ClientLogger

Description: Starting the logging utility in a client machine.

Precondition: Client Discovery phase is completed AND the current machine ID is a Client ID AND if remote logging is enabled the DirectorLogger RMI is running.

Course of events:

- Alice initializes the ClientLogger singleton with the machine ID and the Director's ID if remote logging is allowed.
- Alice configures the logging options using a Preferences node **OR** manually.

- Alice runs the instance of ClientLogger.
- The ClientLogger starts running.

Postcondition: The ClientLogger singleton is up and running waiting for log instructions from the ClientWrapper associated with its machine.

Bootstrap: Before starting the ClientWrapper implementation, the Bootstrap initializes the ClientLogger singleton and configures it using a Preferences node. If remote logging is enabled it also configures the Director ID of the machine where the logger must send log requests.

1.1.4 The Monitor

Use Case: Starting the Monitor.
Description: Start a monitoring utility that will supervise the execution based on the logs collected by the DirectorLogger.
Preconditions: Client Discovery phase is completed AND the current machine ID is the Director's ID AND remote logging is enabled and running AND monitoring is enabled.
Course of events:

OPTION 1: Manual configuration and startup.

- Alice creates a Monitor with the Director's ID, the DirectorLogger.
- Alice creates and associates an OutputParser with the Monitor.
- Alice sets the necessary configuration parameters.
- Alice runs the Monitor.

OPTION 2: Configuration and startup using a Preferences node.

- Alice creates a Monitor with the Director's ID, the DirectorLogger and a Preferences node.
- Alice runs the Monitor.

Postcondition: The Monitor is running and will periodically poll data from the DirectorLogger for evaluation, then process the data with the OutputParser.

Bootstrap: Bootstrap configures the monitor using a Preferences node. It then runs it after running the DirectorLogger, only if both remote logging and monitoring are enabled.

1.2 Interactions

We have seen how Alice (or the Bootstrap) sets up the necessary components to run an experiment. From this point on the Tester runs until there is no more load to emulate. We will now expose several Use Cases that do not involve Alice as an actor but that describe the component interactions we found interesting. In this subsection we will outline the load generation of session tasks, their scheduling and their process by a TPCWSessionClientWrapper.

1.2.1 Load Generation and session injection

Use Case: Generation of a set of delays.

Description: Generate a portion of requests represented by delays.

Actors: StandardLoadGenerator, StandardFileInputDelayList, DelayCalculator

Precondition: The StandardLoadGenerator and the StandardFileInputDelayList have been initialized AND The input file is a csv file containing timestamps and number of requests.

Course of events:

- Upon creation the StandardFileInputDelayList scans the input file and reads the first pair timestamp, number of requests.
- For each pair it scales both values using configuration scaling factors and determines the interval between timestamps.
- Once the data is scaled, it requests the Singleton DelayCalculator to sparse the number of requests in the given interval, obtaining a list of delays between requests.
- The StandardLoadGenerator polls the DelayList for a portion of the delays.

Postcondition: The StandardLoadGenerator obtains a sublist of delays representing a relative schedule of requests.

Use Case: Session ID injection

Description: Creating a list of LTSessionTasks based on a list of delays.

Actors: StandardLoadGenerator, SessionInjector

Precondition: UseCase:Generation of a set of delays AND The SessionInjector has been configured.

Course of events:

- The StandardLoadGenerator requests a delay list to be processed.
- The InputProcessor implementation SessionInjector generates a list of Session Ids (SID) using configuration parameters.
- The SessionInjectors associates a random number of requests per SID.
- The SessionInjector associates each SID to a delay in a round robin fashion, creating LTSessionTasks.
- When a SID has been assigned to its maximum number of requests the SessionInjector injects an end of session SID to the last delay.

Postcondition: The StandardLoadGenerator obtains a list of LTSessionTasks representing a relative schedule of requests associated with their SID.

1.2.2 Session scheduling

Use Case: Scheduling LTSessionTasks

Description: Distribution of session-based load.

Actors: LoadDirector, SessionScheduler

Precondition: The SessionScheduler has been initiated and configured AND The LoadDirector has obtained a list of LTSessionTasks from the StandardLoadGenerator.

Course of events:

- The LoadDirector requests a distribution of the LTTasks indicating the number of schedules it wishes the load to be split in.
- The SessionScheduler keeps track of the association Client-SIDs.
- The SessionScheduler distributes LTSessionTasks among clients adjusting their relative delay.
- When an end of session is scheduled the SessionScheduler removes the association Client-SID.

Postcondition: The LoadDirector obtains one schedule of LTSessionTasks for each client. The original schedule has been split focusing on a parallel execution of the load, taking in consideration that all requests for a specific session must be sent to the same client.

1.2.3 Executing session tasks

Use Case: Sending and receiving a schedule.

Description: The schedule is sent to the appropriate client and processed.

Actors: LoadDirector, ClientWrapper

Precondition: The ClientWrapper is running and is accessible by the LoadDirector.

Course of events:

- The LoadDirector sends a schedule of requests to the ClientWrapper via RMI.
- The ClientWrapper enqueues the requests using synchronization to avoid unconsistency.
- The LoadDirector waits for a given time and the process is repeated.

Postcondition: The ClientWrapper obtains a list of requests to execute.

Comments: The ClientWrapper can accept and enqueue schedules but does not interact with their data unless it has been initiated. Initiation is done by the LoadDirector (or Alice) by setting a specific initiation time to indicate the beginning of the client's activity.

Use Case: Processing an LTSessionTask schedule.

Description: Interpretation and preparation of the LTSessionTasks received at the client.

Actors: TPCWSessionClientWrapper

Precondition: The schedule is a list of LTSessionTasks AND The TPCWSessionClientWrapper is running and initiated (initiation time < current time)

Course of events:

- The TPCWSessionClientWrapper dequeues the first LTSessionTask and extracts its SID.
- The TPCWSessionClientWrapper keeps track of the EB[1]-SID association.

[1] TPCW Emulated Browser, see Section 3: Related Work, subsection 3.8: TPCW.

- If no active EB is assigned to the SID, the TPCWSEssionClientWrapper waits *delay* milliseconds and activates a new EB associating the SID to it.
- If an active EB is assigned to the SID it extracts the delay of the LTSessionTask and assigns it to the EB.
- If an end of session is detected the TPCWSessionClientWrapper will inform the EB.
- Once the LTSEssionTask is assigned the TPCWSessionClientWrapper waits for a given time and dequeues the next task.
- Throughput is periodically logged at the ClientLogger.

Postcondition: All tasks are assigned to their respective EBs. The TPCWSessionClientWrapper waits for more schedules.

Use Case: Session Execution at the EB.

Description: For each session, one EB is responsible of sending requests based on the delays received from the TPCWSEssionClientWrapper.

Actors: EB

Precondition: The EB has been activated AND The EB is configured to use sessions.

Course of events:

- Upon activation, the EB sends its first request for the homepage of the TPCW Servlet.
- After receiving a delay, the EB waits and sends the next request to the URL obtained using a Markov chain.
- Once it receives data from the service he EB logs the latency using the ClientLogger.
- When an EB receives an end of session signal it completes its requests and terminates execution.

Postcondition: All requests of a given session have been sent using a TPCW Emulated Browser.

1.3 Configurations and extensions

We have exposed several use cases representing some of the usage possibilities of the Tester. Depending on how Alice configures it, the Tester will have a particular behavior, additionally it can be modified and extended. In the following lines we will list and describe several of the alternative usages of the Tester, more detailed instructions can be found in the User Manual.

1.3.1 Existing configuration possibilities

Constant load generation: Alice can run the Tester without using a custom input file by specifying a list of rates (Requests per second) and their respective duration.

Custom Input: The utility InputPattern has been designed to extract values of any type of input provided Alice configures it to *digest* the specific input.

Different client options: We have seen a TPCW session based ClientWrapper. We have implemented two other options as specified in the Architecture section. The StandardClientWrapper executes a ServiceClient request for each *delay* received, this is the basic client and can be customized to use any type of service. The TPCWClientWrapper executes an EB for each *delay* received.

Scaling parameters: several configuration parameters can be scaled affecting the load, the monitoring intervals, the number of concurrent sessions etc. (see manual)

1.3.2 Replaceable and extensible modules

The tester uses several interfaces to allow for custom implementations, Alice can for example implement her own DelayList but use it with the existing StandardLoadGenerator and a custom InputProcessor. We present a list of the replaceable components:

LoadGenerator: as a bridge between the delay list and the LoadDirector.
DelayList: as a bridge between the delays and the LoadGenerator.
InputProcessor: to alter the delays and create LTTasks.
LTTask: to wrap data associated to each request.
Scheduler: for custom load distribution.
ClientWrapper: for custom load execution.
ServiceClient: to execute the service-specific request.
DirectorLogger: for postprocessing of output data.
Monitor: for additional monitoring or *live* load modification.
OutputParser: for a custom data extraction from logs.

Figure 5.2 : The Tester: Interfaces to the key components.

Additionally, all components can be extended, for example Alice can use the LoadDirector's methods and variables but modify its execution. As shown previously, all Bootstrap responsibilities can be equally bypassed by not executing it and only using its methods. For example, a GUI could be implemented and use the Bootstrap as a factory for the creation and configuration of all components.

1.3.3 Example

As a broader example, we propose an advanced user configuration and extension for Alice to implement. This is only one possibile setup, we have intended to illustrate the variety of modifications therefore it is relatively complex and requires additional design, suitable for a project of small proportions.

Unchanged components:

LTBootstrap: using only its methods in manual mode.

StandardLoadGenerator

DelayCalculator

ClientLogger

Extended and configured components:

LoadDirector: extended to use a pool of clients that allows the monitor to request client dropping and recycling.

InputPattern: configured to accept and extract data from the following input pattern:

User ID: uid	Session id: sid	Service Name: sname
Date: date	Time: time	Session Duration : duration
Request id: rid	Request text: rtext	Request frequency: rfreq

Custom implementations:

Component NAME	***Interface***	***Custom properties***
UserSessionFileInputDelayList	DelayList	Requests the *date*, *time*, *rfreq* and *duration* from the InputPattern and generates all delays for an input entry.
UserSessionInputProcessor	InputProcessor	Requests additional data (uid, sid, sname, etc.)from the InputPattern and associates it with each delay creating LTUserSessionTasks.
LTUserSessionTask	LTTask	Wraps delays and additional data
UserSessionScheduler	Scheduler	Schedules the LTUSerSessionTasks distributing them among clients assigning a specific LTUserSessionTask to a client depending on the *sname* of the service. Assuming some clients execute different services.
UserSessionClientWrapper	ClientWrapper	Uses a specific serviceClient *sname.* For each LTUserSessionTask, it sets the rid and rtext in the serviceClient and executes it.
*sname*ServiceClient	ServiceClient	Executes different types of requests for *sname* , identified by rid and sends *rtext* to the server. Logs data on the uid, sid, rid, response and latency.
UserSessionDirectorLogger	DirectorLogger	Logs every received data to the file system but builds a temporary dynamic log discarding log messages containing information on a specific *rid.* This log is used by the UserSessionMonitor.
UserSessionMonitor	Monitor	Generates histograms based on the last *t* minutes of activity. Requests the LoadDirector to drop a

Component NAME	*Interface*	*Custom properties*
		client or recycle it depending on the client's performance.
UserSessionOutputParser	OutputParser	Utility that assists the Monitor parsing the DirectorLogger logs and extracting specific data.

Figure 5.3 : The Tester: Example of a custom user implementation of our interfaces, definition of the classes.

For this example we have chosen a complex scenario to illustrate the different needs users might have when testing a given service. In this regard the Tester offers a design structure that allows for combination of standard and custom implementations of its components.

2 Ideas, Diagrams and illustrations

In this subsection our purpose is to offer a clear view of the project illustrating all previous descriptions using both class and sequence diagrams. We will begin with a class overview depicting the relations between components, followed by a closer look into attributes, methods and design patterns to finally represent some of the above specified use cases in sequence diagrams. Even though this is a Design document, in the following lines we will begin to mention aspects of the implementation.

2.1 Design Overview

The Tester comprises 10 interfaces, 24 implemented classes including 7 utilities. An interface is an abstraction of a component that specifies its interactions with other components, all implementations must guarantee the preconditions and postconditions of every method in the interface to guarantee consistency of usage. Each implementation we offer represents a possible course of action to fulfill the interface's specification, additionally different combinations of implementations lead to different scenarios as shown previously in this document. We call *utility* a standalone component with a specific function, that we have disassociated from the core for design purposes but whose methods could have been integrated in one of the core components. Figure 4.1 depicts the class diagram of our system, in the following lines we will briefly comment it in relation to all the concepts we have exposed previously. The purpose of this evaluation is to not only to clarify the roles of each component, but to offer a critical overview of our design.

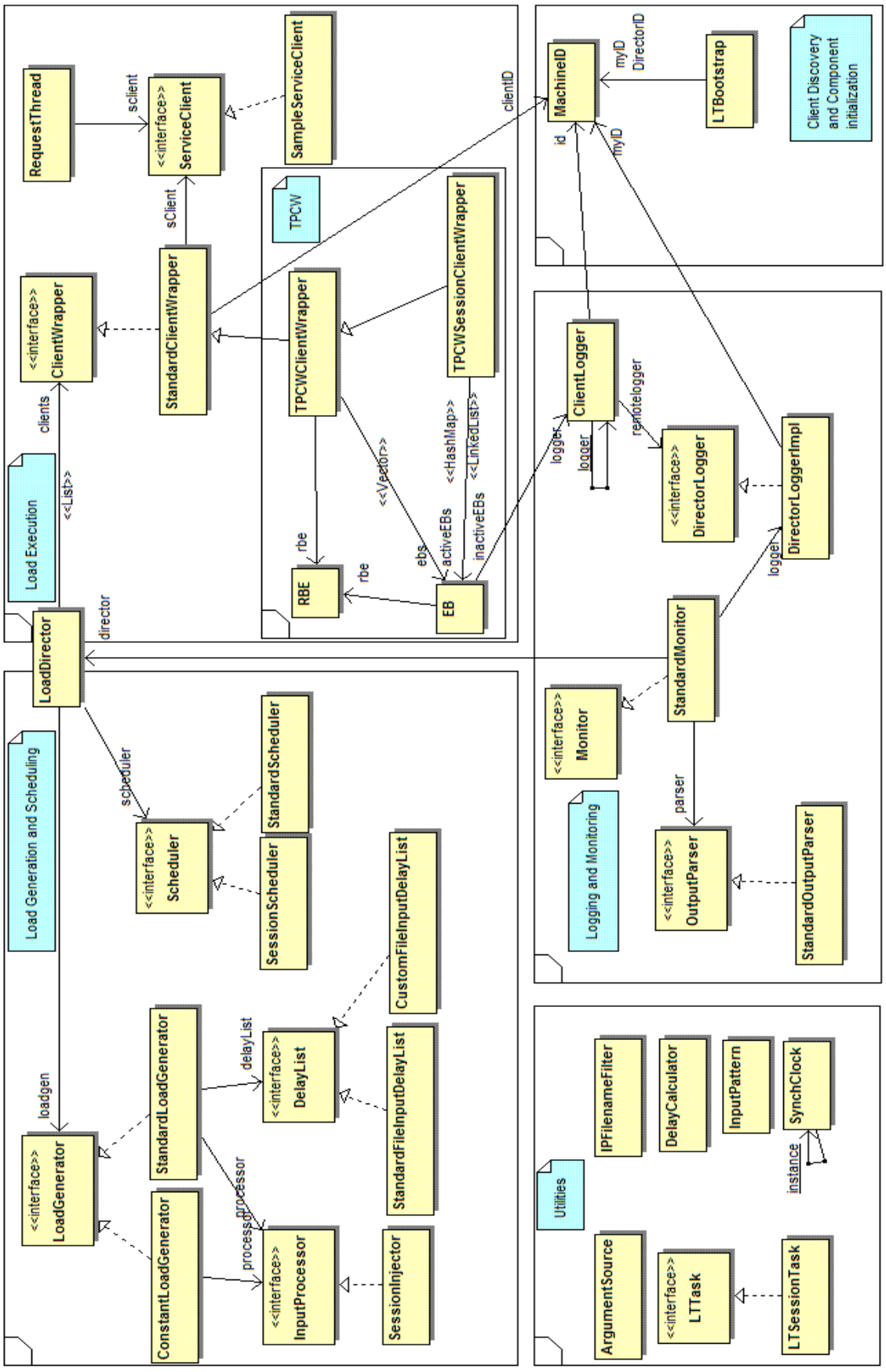


Figure 5.4 : The Tester: Global view of the Classes.

2.1.1 Observations

An Interface-based architecture.

The first detail we can notice is the importance of interfaces, the LoadDirector -which is what we could call the center and brain of our design- is only connected to interfaces allowing for a custom plug-in usage of our system. The three interfaces related to the LoadDirector illustrate a minimalistic design to fulfill the purpose of the Tester: collect load (LoadGenerator), distribute it (Scheduler) and emulate it (ClientWrapper).

Excessive layering: Poltergeist vs God Complex antipatterns

On the other hand we can observe four different actors in the load generation process (including the LoadDirector), separating functionalities in different layers. This choice of design adds a seemingly unnecessary complexity. The *Poltergeist* antipattern points out the flaws of using objects exclusively to pass information to other objects. In this case the LoadDirector, an intermediate between the load producers and consumers, is an example of a *Poltergeist*.

The file input based load generation is split between the StandardLoadGenerator, the StandardFileInputDelayList and an InputProcessor. A different approach merging functionalities would lead to a better understanding of the load generation process. For example, the LoadDirector could bypass the StandardLoadGenerator and access delays directly from the DelayList, eventually processing them at the InputProcessor if necessary. This approach grants a more compact and focused design, nonetheless it would give much more responsibility to the LoadDirector leading to a *God Complex* antipattern. When designing the load generation components we considered the following premises:

- The DelayList is only an *Iterator* through delays and only needs to know how to generate delays from an input.
- The LoadGenerator acts as a *Bridge* between the input and the load processing, delivering to the LoadDirector data without knowing its structure, format or purpose.
- The InputProcessor is an LTTask *Factory* and can be considered as a utility that has been decoupled from other components for clarity purposes.
- The LoadDirector requests load and does not need to know how it is obtained nor its structure. The LoadDirector is, as a matter of fact, one of the expendable components of our system.

Is the LoadDirector useless?

The reason why the LoadDirector is not an interface itself is because it does not necessarily have to be part of the system, it uses all other components and orchestrates the test which makes it a good candidate for a GUI substitution.
By looking at the LoadDirector we can determine how the system works, all we need is a LoadGenerator, a list of ClientWrappers and a Scheduler. As a matter of fact, a simplified version of our system could have been designed as follows:

1. Alice runs a script using the LoadGenerator to extract load from the input file and storing it into another file.
2. Alice runs a script using the Scheduler to distribute the load from the load data file into several files representing schedules.
3. Alice manually starts a client in every machine with their schedules.
4. The clients run and dump output data to disk.
5. Alice evaluates and parses the output data to generate traces and diagnosis.

This implementation example underlines the idea that there are several approaches to the purpose of this project, and the one we have chosen is not the least complex one. Among many secondary components it discards three of the main ones: the BootStrap, the LoadDirector and the Monitor. In the following lines we will expose our reasons for implementing them and their value in the Tester.

2.1.2 Expendable Components

An important fact that justifies our choice of design is the distributed nature of our system. The Tester must run in several machines simultaneously, configure each execution and establish communication between them. Several benchmarking tools require for a specific platform and network configurations to allow for a central machine that would act as the Director to start the execution of Clients on other machines. Other implementations require for the users to start all the Client machines manually and specify their Location to the Director machine. Our purpose was to reduce the user responsibility while keeping the Tester environment-independent.

Biology and Computer Science.

The idea behind the LTBootstrap class derives from the concept of cellular differentiation[1], which is the process by which less specialized cells such as stem cells modify their type depending on the active information they carry. This process is deterministic, which is an interesting feature to grant a predictable behavior in a software execution. Embryonic cells contain all information necessary to become any other type of cell in the same way that the LTBootstrap is created using a Preferences node with the same configuration for all its instances. We needed to find a differentiation trigger capable of uniquely identifying a machine, which led to the idea of a machine ID derived from the IP address. This feature delegates the user responsibility of locating and starting all machines to an automatic process of discovery. Once the LTBootstrap determines the role of the machine, it technically becomes it by running either the LoadDirector or the ClientWrapper implementation on its thread. Combining this characteristic with a shared file system where all instances of LTBootstrap can identify themselves by writing their ID we obtain a mechanism to run the Tester before knowing *where* it will run. The Tester could be extended for example to run in a wider distributed system (with a Dynamic Hash Table for example), accepting Client churn and adapting itself to redistribute the load depending on the amount of clients available. Which leads to the need for control of load distribution depending on the amount of clients.

A brain.

The LoadDirector is responsible of controlling the client activity and is the only component that communicates with clients. In our implementation, the Scheduler is initially configured to split the load between a fixed number of clients, assuming the number will not vary. Nevertheless it accepts instructions to drop and add clients from the LoadDirector which might be necessary to keep load distribution consistency in case of a client failure.

Additionally when the LoadDirector runs, it automates the load collection and delivery according to configuration parameters, its main responsibility is to keep a balance between load generation overhead and network overhead.

A second brain.

One of the latest additions to the project has been the remote logging and monitoring classes. Our purpose has been to allow for a constant feedback to the Director JVM on the evolution of both the client-perceived latency, throughput and errors. The latest version of the Tester has been closed with a monitor capable of collecting data from logs and obtaining information on general

[1] However not very Computer Scientist-like, this was the concept that led to the design, and not a design pattern.

behavior as well as client-specific behavior. The idea behind this feature is to allow the monitor to decide, depending on its configuration, whether there is a need to modify the current scenario. The monitor would be capable of notifying which clients should be dropped on account of their performance, or whether the load should be modified on account of the latency.

A future interface

Both the Monitor and the LoadDirector are candidates to be merged into an GUI providing control and supervision over the client execution. The current design targets an execution in a cluster, scheduling the launch of a given number of LTBootstrap to collect data once the experiment is finished. Nevertheless our design allows for more user interactive extensions that we will comment in Section 8 (Conclusions).

We have exposed the reasons behind our choice of design, focusing on the need for independence between components. A different scenario might need the input collection to be executed on one or several machines using a remote DelayList service, or the possibility of requesting load execution from several LoadDirectors. We have focused our decisions on making different approaches possible with only changing a small number of components, by either creating new implementations or extending the existing ones. The LTBootstrap, LoadDirector and Monitor classes are expendable to allow for an easier refactoring of the startup, control and decision making processes.

2.1.3 Design Patterns and solutions

Scenario: We had the need to synchronize all clients with the director and decided that the LoadDirector would periodically send each ClientWrapper the value of its current offset.
Problem: In the client machine, not only the ClientWrapper needs to know the global time, for logging purposes the ServiceClient and the EB also need this value.
Solution: We used the Singleton Pattern and created the SynchClock, an object that with only one instance which can be obtained statically.
Comment: The same pattern has been applied to design the ClientLogger which needs to be accessed by all the components of the Client.

Figure 5.5 : Singleton Pattern

Scenario: We were requested to extract data from file input and generate load.
Problem: We needed to decouple the LoadGenerator from the structure of the input.
Solution: The DelayList combines the Iterator pattern offering methods to iterate through the list of delays, and a simplified Builder pattern. The Builder pattern provides the abstraction necessary to allow for custom delay generation through the DelayList interface. When the LoadGenerator requests for a portion of the delays he does not need to know whether those delays are generated upon instantiation, upon initialization or upon request (using the DelayList.next() method).

Figure 5.6 : Iterator and Builder patterns

Scenario: We adapted the ClientWrapper to be able to start Emulated Browsers from the TPCW java implementation in order to perform the TPCW benchmark.
Problem: The TPCW java implementation requires users to specify the number of EBs it will use before runtime, whereas our implementation will start one EB per session. We then faced the problem of running out of EBs before we run out of sessions.
Solution: Using the Object Pool pattern we recycle EB objects once they have completed their session. The sessionTPCWClientWrapper contains information about the active EBs which are running and the inactive EBs which can be restarted with a new session.

Figure 5.7 : Object Pool pattern

Scenario: We have created two ClientWrapper implementations to work with TPCW. The TPCW java implementation uses as main class an RBE, as mentioned before several factories and key components of its architecture reference to both the class and the instace of RBE.
Problem: We use implementations of ClientWrapper which are also subclasses of a remote object to allow for the RMI service, therefore due to Java inheritance restrictions they can't also be a subclass of the RBE.
Solution: We have applied the Composite Pattern to allow for multiple inheritance by adding an instance of RBE as an attribute in the ClientWrapper TPCW implementations. All calls to TPCW utilities and factories will contain this instance as if the caller was the RBE.

Figure 5. 8 : Composite pattern

2.2 A closer look

We will now present a more detailed description of the classes we have implemented, we also add a representation of the ones we have considered interesting. Some descriptions are more detailed than others to allow for a better understanding of how the class is used. All figures shown in this subsection have the following format:

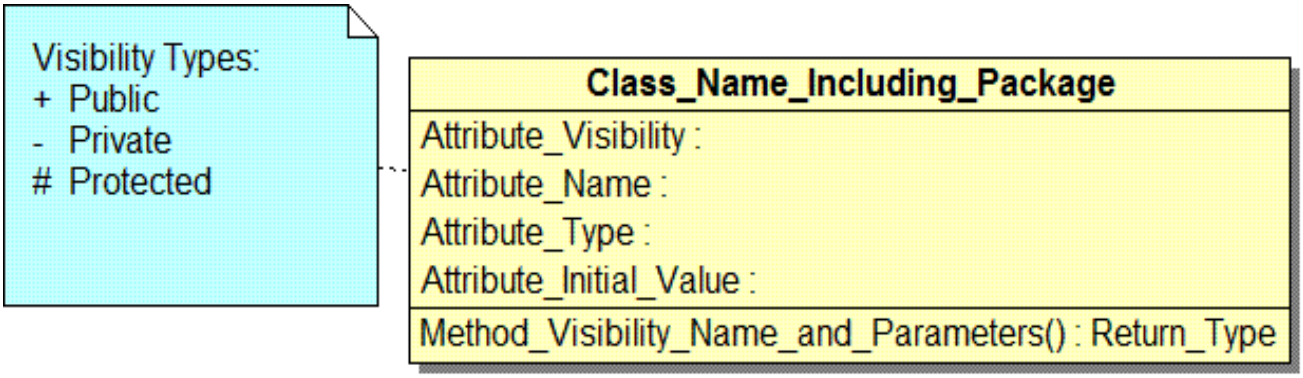


Figure 5.9 : Contents of the representation of a class

2.2.1 The Director

The LoadDirector

Purpose: location of clients and control of the load.

Attributes:

- *synchInterval:* interval between clock synchronizations.
- *intervalLength:* size in milliseconds of each interval of load requested from the LoadGenerator.
- *dataProcessingLatency:* estimated value of the amount of time necessary to generate an interval.
- *serviceName:* classname (package.classname) of the ServiceClient class we are using.
- *currentInterval:* interval of requests being processed.
- *distributedRequests:* list of the schedules that will be sent to the clients.

Methods:

- *getClients:* obtains the remote objects by looking up the client IDs
- *init:* initializes the LoadDirector with a preferences node.
- *initializeClients:*sets the time when clients will begin executing their schedules
- *run:* execution of the loadDirector to periodically poll a schedule, distribute it and send it to the clients
- *stopClients*: *stops all clients*
- *synchronize:* computes the offset between the Director's and the Client's System time and sends it to the client to synchronize it with the local time
- *synchronizeAll:* synchronizes all clients
- *updateRequests:* sends each client a new schedule

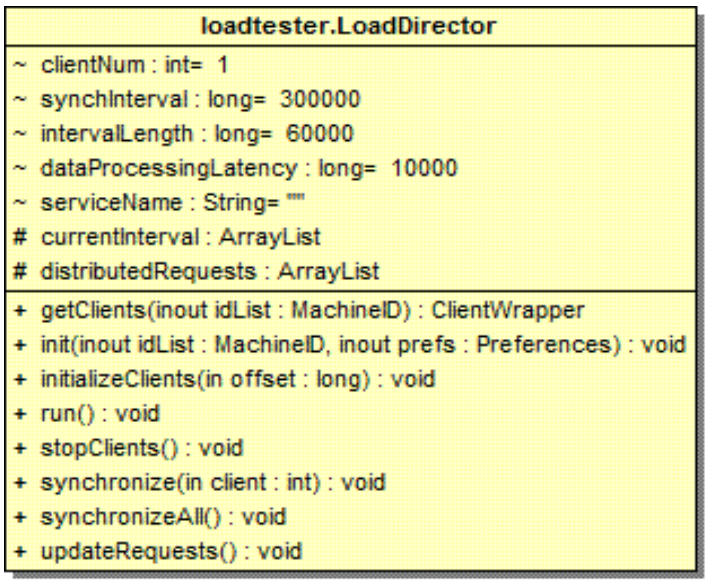


Figure 5.10 : The LoadDirector class representation

Creator: LTBootstrap or Alice

Uses: Scheduler, LoadGenerator, ClientWrapper.

The StandardLoadGenerator

Purpose: load and task generation

Features:

Upon request,

- Requests delays from the DelayList
- Processes delays with the OutputParser
- Returns a schedule of requests

Used by: LoadDirector

Uses: DelayList, OutputParser

Related Components: none

The StandardFileInputDelayList

Purpose: load extraction from input file.

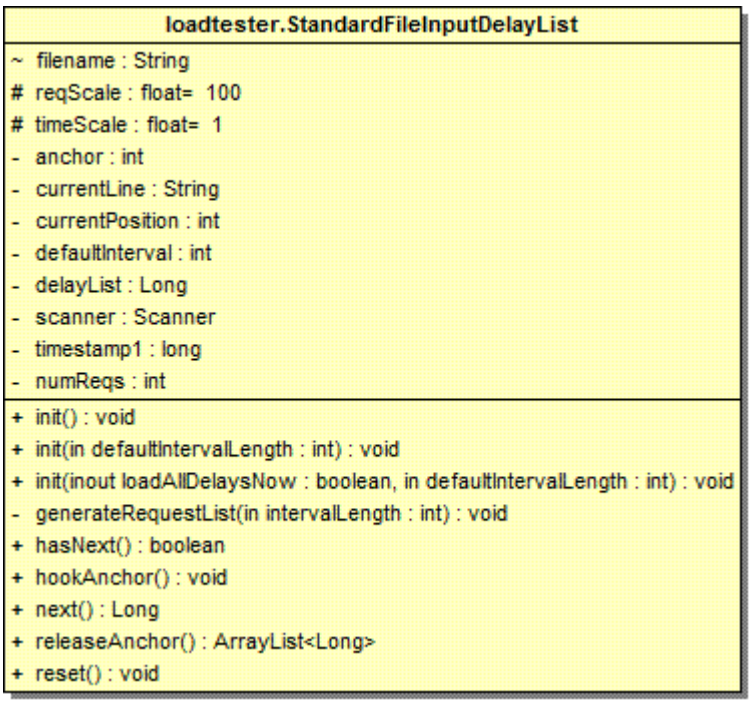


Figure 5.11 : The StandardFileInputDelayList class representation

Implements: DelayList

Attributes:

- *filename*: name of the input file
- *reqScale*: value by which the number of requests are scaled
- *timeScale*: value by which the time is scaled
- *anchor*: checkpoint marking an initial position in the list for sublist extraction. In this implementation it is an integer since the list is generated upon creation and not dynamically
- *currentLine*: current line being read from the input
- *currentPosition*: current position of iteration
- *defaultInterval*: default length of the interval extracted from input
- *delayList*: internal list of delays
- *scanner*: Java scanner to iterate through a file
- *timestamp1*: reference to the current timestamp being read
- *numReqs*: reference to the number of requests being read

methods:

- *init*: different initialization methods
- *generateRequestList*: appends an interval of intervalLength to the delayList
- *hasNext* and *next* methods for iteration
- *hookAnchor* and *releaseAnchor* methods to extract a sublist
- *reset*: restart the iteration from the initial point

Used by: LoadGenerator

Uses: DelayCalculator

The SessionInjector

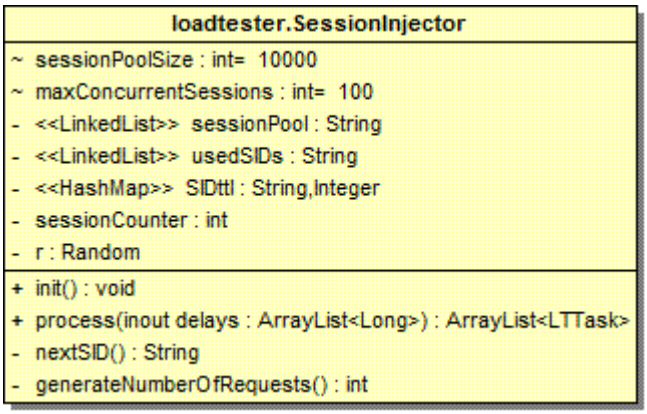


Figure 5.12 : The SessionInjector class representation

Purpose: association of session ids with the load.

Attributes:

- *sessionPoolSize*: maximum number of session ID generated.
- *maxConcurrentSessions*: maximum number of concurrent sessions
- *sessionPool*: list of the available SID
- *usedSIDs*: SIDs being currently used
- *SIDttl*: relation of SID and the number of requests remaining for that SID
- *sessionCounter*: number of active sessions
- *r*: random generator for the number of requests per SID

Methods:

- *init*: initializes the attributes
- *process*: generates a list of LTSessionTasks by adding an SID to each delay
- *nextSID*: returns the next available SID, updates all necessary attributes, returns negative SID to represent an end of session
- *generateNumberOfRequests*: randomly generates the number of requests for an SID

Used by: LoadGenerator

Uses: LTSessionTask

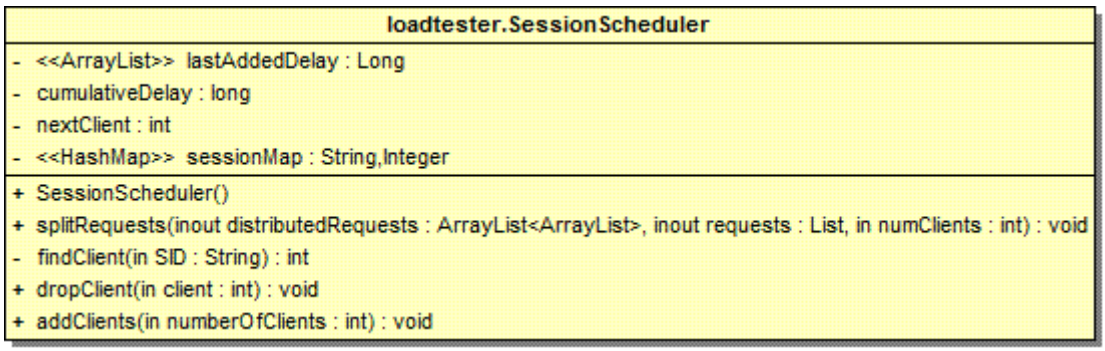


Figure 5.13 : The SessionScheduler class representation

The SessionScheduler

Purpose: consistent distribution of session-based load.

Attributes:

- lastAddedDelay: list of the last delay added to each client schedule, used to distribute delays from one schedule to many adjusting the relative time
- cumulativeDelay: for a given client schedule, it is the delay cumulated since the last added delay
- nextClient: client to whom the next request will be assigned to
- sessionMap: relation of SID and clients to keep track where requests of a specific SID should be assigned to.

Methods:

- splitRequests: splits a list of requests among a number of clients returning a list of distributed requests.
- findClient: finds the target client where the request with a given SID should be sent to.
- dropClient: removes a client from the schedule targets
- addClient: adds a client to the schedule targets

Used by: LoadDirector

Uses: LTSessionTask

The StandardMonitor

Purpose: evaluation of metrics.

Features:

- Periodically polls log data from the DirectorLogger
- Requests metric extraction at the OutputParser
- Prints information on:
  - average latency
  - maximum latency
  - number of errors
  - throughput
  - average latency per client
  - number of errors per client

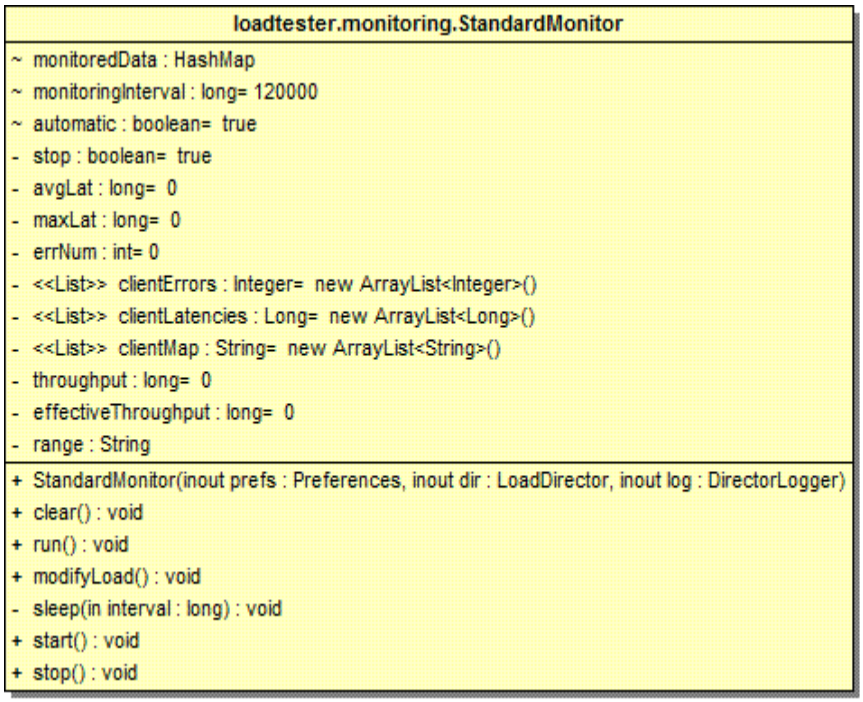


Figure 5.14 : The StandardMonitor class representation

Created by: LTBootstrap, Alice

*Uses:*DirectorLogger, OutputParser

The StandardOutputParser

Purpose: extraction of metrics from logs.

Features:

- Upon request processes a list of log messages and extracts data based on the type of message (LAT: latency, THR:throughput, ERR: error, INFO: other)
- The metrics can then be polled
- Every call to processData will overwrite the metrics

Used by: Monitor

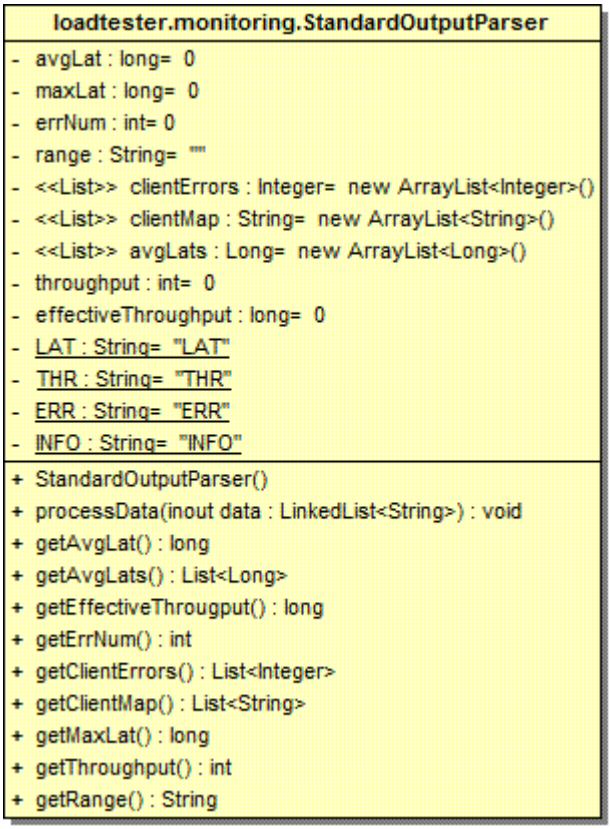


Figure 5.15 : The StandardOutputParser class representation

The DirectorLogger

Purpose: Remotely log all clients output data.

Features:

- Runs as an RMI service
- Accepts logs from remote clients using a linkedblocking *messageQueue*
- Keeps a portion of the log of size =*inMemoryLogSize* in the *dataList*
- Periodically dumps a part of the dataList to a file specified by *logFileBasePath/logFileBaseName* or *LT_HOME/LogFileBaseName* if no path is specified
- Upon request, returns the first *numMessages* messages of the dataList.

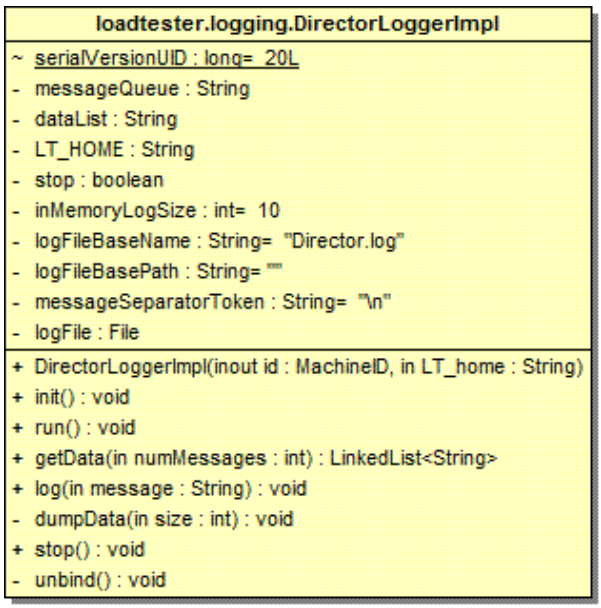


Figure 5.16 : The DirectorLogger implementation class representation

Created by: LTBootstrap, or Alice

Used by: ClientLogger, Monitor

2.2.2 The Client

The StandardClientWrapper

Purpose: execution of a load schedule of delays using a *ServiceClient* wrapped in a *RequestThread*.

Implements: ClientWrapper

Features:

- Runs as an RMI service
- Accepts Updates of the delays by adding them to the delayQueue
- For each delay, executes a *ServiceClient* wrapped in a *RequestThread*
- Accepts synchronization messages and updates the *SynchClock*
- Periodically logs information on the throughput

Created by: LTBootStrap, or Alice

Uses: ServiceClient, RequestThread, ClientLogger, SynchClock

Used by: LoadDirector

The TPCWClientWrapper

Purpose: execution of a load schedule of delays using TPCW Emulated Browsers

Extends: StandardClientWrapper

Features:

- Inherits all features from the StandardClientWrapper
- Instead of launching a RequestThread /ServiceClient per delay, it starts an Emulated Browser.

Used by: LoadDirector

Uses: SynchClock, EB, RBE, ClientLogger

The TPCWSessionClientWrapper

Purpose: execution of a load schedule of LTSessionTasks using TPCW Emulated Browsers

Extends: TPCWClientWrapper

Attributes:

- *serialVersionUID:* a class-specific identifier to indicate how a serialized object must be unserialized. (Objects must be serialized before being transported through the network using RMI, to be unserialized they must contain information on their original class)
- *SessionMap:* Used to assign session IDs to Emulated Browsers

Relations:

- *ActiveEBs:* list of EBs currently executing a session
- *InactiveEBs:* pool of unused EBs

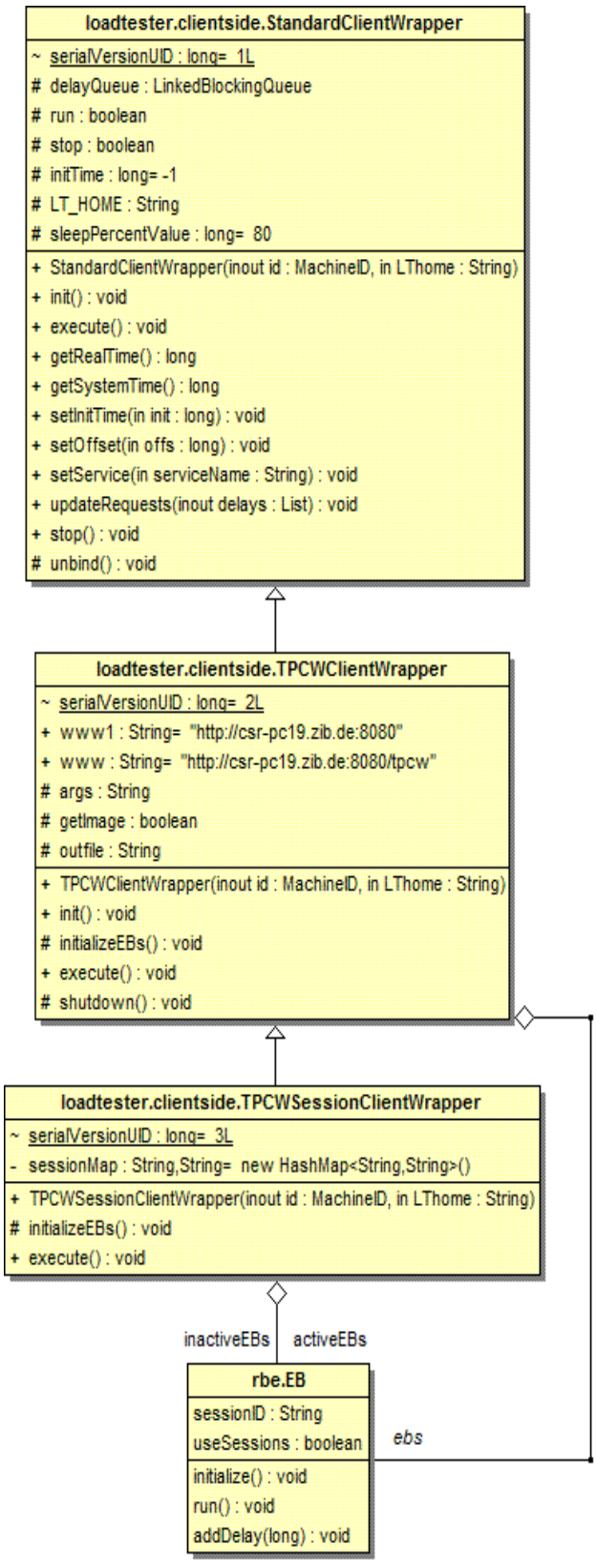


Figure 5.17 : The Three ClientWrapper Implementations class representation

Features:

- Extends all features of the TPCWClientWrapper
- Parses LTSessionTasks and assigns delays to the appropriate EB
- Recycles EBs
- Detects the end of a session and communicates it to the appropriate EB

Used by: LoadDirector

Uses: SynchClock, EB, RBE, ClientLogger

Comments: On Figure 5.17 we have shown the three ClientWrapper implementations along with the associated EB to the two TPCW ones. The class representation for this EB has been modified to hide most of its attributes and methods, leaving only the ones we have added or changed to allow for our TPCW Session plugin.

The ClientLogger

Purpose: Log client output data on demand

Attributes:

- *messageQueue:*queue of logged messages.
- *stop*: whether the ClientLogger should stop running
- *logFile*: name of the log file
- *numberOfMessagesPerTrans:* size of the set of messages periodically sent to the DirectorLogger
- *baseFrequencyOfMessages*: estimated frequency of message log requests to the ClientLogger

Figure 5.18 : The ClientLogger class representation

- *logFileBaseName*: prefix to the log file, *logFileBasePath*: path to the log file
- *messageSeparatorToken*: token used to separate different messages when we wrap several messages in a String for remote logging. This value must be identical in all loggers.

(methods not described)

Used by: ClientWrapper, EB, RBE, ServiceClient, RequestThread

Uses: DirectorLogger

2.2.3 Utilities

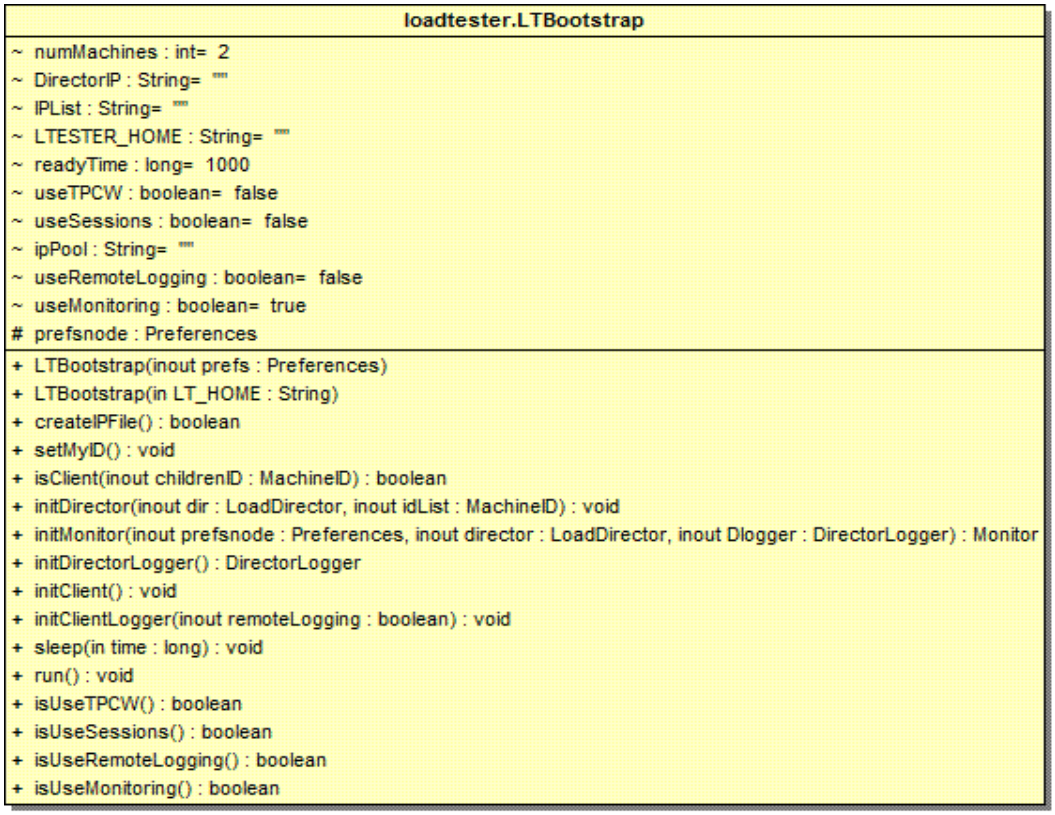


Figure 5.19 : The LTBootstrap class representation

The LTBootstrap

Purpose: Instantiate Client components in the Client JVMs and Director components in the Director JVM.

Attributes:

- *numMachines*: number of JVMs running LTBootstrap
- *DirectorIP*: IP of the Director JVM (optional for when users want to force a Director machine)
- *IPList*: list of the IPs of all JVMs (optional for when users want to bypass client discovery)
- *LTESTER_HOME*: path to the installation of the Tester.
- *ReadyTime*: estimated time necessary to have a Client RMI running (optional but reccomended in case of slow network or slow machines)
- *ipPool*: path to the shared file system folder where clients can write their IPs for client discovery. (optional for when we want to use a shared file system for client discovery, obligatory if no IPList is specified)

(no methods described, view figure 5.21 for a sequence diagram of client discovery)

The DelayCalculator

Purpose: Singleton used to sparse requests in a time interval given the request rate for the given interval.

Features:

- Uses a negative exponential distribution to split a given length of time in a specific number of portions of different sizes representing the delays between the requests that have been sent in an interval.

Used by: DelayList, ConstantLoadGenerator

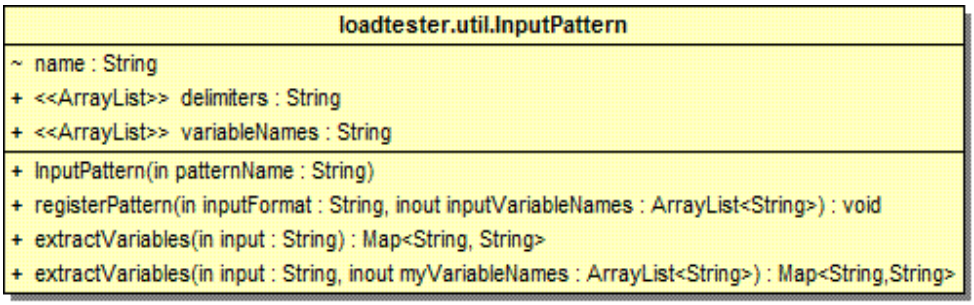


Figure 5.20 : The InputPattern class representation

The InputPattern

Purpose: Singleton used to extract input data from input text using token recognition. Similar to java.util.regex.Pattern. This class stores a string pattern containing constant and variable tokens.

Attributes:

- *name*: name of the pattern
- *delimiters:* list of tokens separating the data we are interested to collect
- *variableNames:* list of the names given by the users to the variables that appear in the input

Methods:

- *registerPattern*: given a template of the input modified by the user in the following manner:
  - All variable content of the input (values that change) is replaced by a variable name designated by the user.
  - A template of the input represents the smallest unit of input that is repeated in the input file.

  The inputPattern generates the delimiter list based on the inputvariableNames thus creating a map of the input.
- *extractVariables*: given a sample of input the InputPattern returns a map of pairs <Variable name, Variable value>. When requested, it will return only a subset of the variables.

2.3 Interactions

We now offer some sequence diagrams to illustrate the client discovery and the load generation, scheduling, distribution and execution. These diagrams are complemented with comments on each step of the flow as well as a brief description. More information can be found in the first part of this design document, at the Use Case Overview.

2.3.1 Client Discovery

We have chosen to illustrate this particular use case because it has been one of the challenges we faced when trying to run the Tester in a cluster where a scheduler leases resources once they are available. Whereas once the cluster job scheduler is ready to run a job it offers the list of the nodes the program will run on -and therefore we could have used this information as input for the tester- we wanted the Tester to be capable of discovering clients without any user input.
Additionally once we added Client IDs as a combination of the IP and a random number to allow for several JVMs to run on the same machine, this feature proved to be useful. This simple method of discovery might allow future extensions of the Tester to dynamically discover node churn and incorporate clients to the execution.

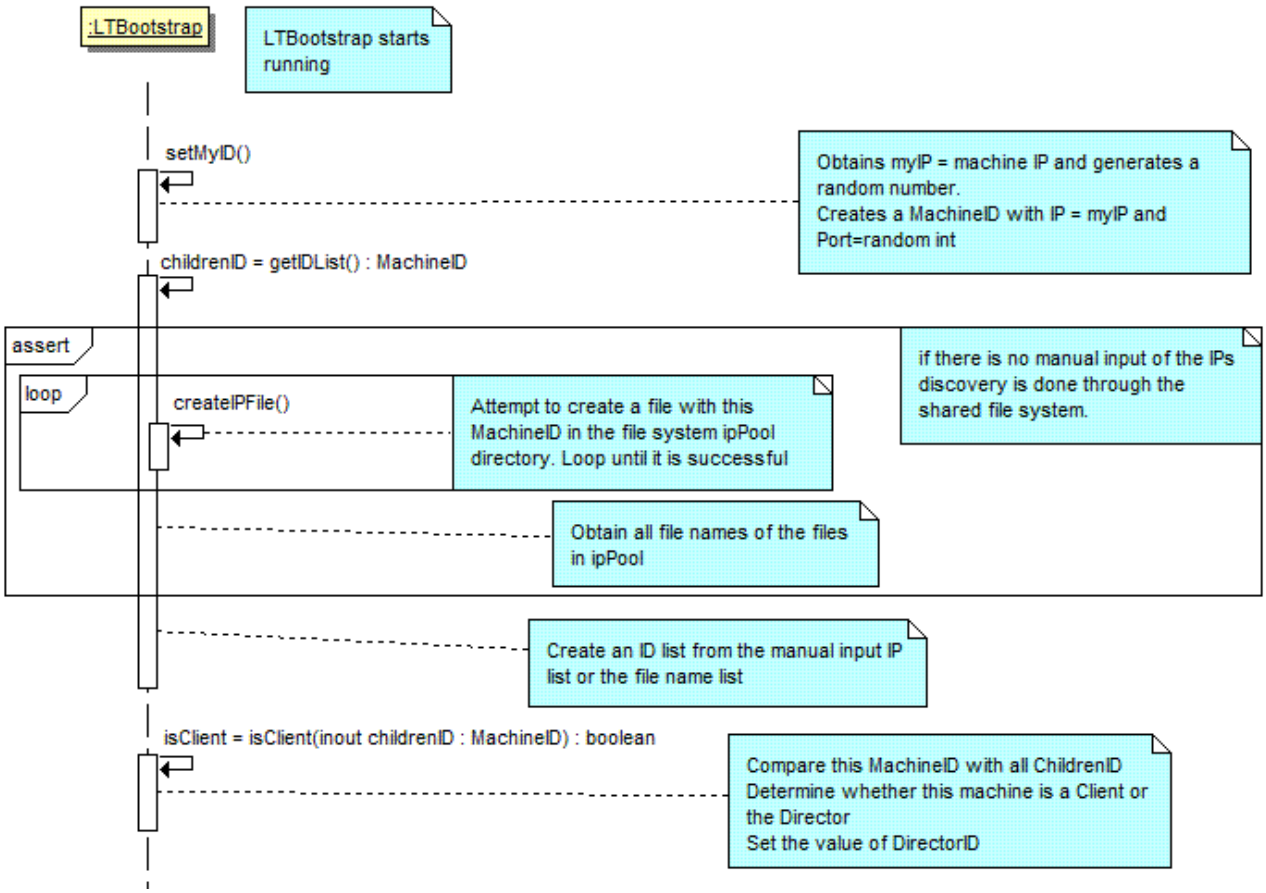


Figure 5.21 : Client Discovery Sequence Diagram

Figure 5.21 shows the sequence diagram corresponding to the process by which the LTBootstrap obtains the IDs of all machines executing the Tester. This operation is much simpler if the user manually inputs the IDs in the configuration file, nevertheless as mentioned before we offer an alternative to the discovery. As we can see, this procedure involves a shared file system where all machines write their identifier. By obtaining all IDs, an instance of LTBootstrap can determine who is the Director, and whether it should be instantiated in this JVM.

2.3.2 Load Generation and Control

The methods by which the Tester generates load are a key element to our design, we have chosen to depict this sequence diagram to illustrate how the LoadDirector and all the components assisting it are configured, and how the load is generated. Additionally this diagram exposes the use of Sminer Preferences, a library we have decided to use to allow for an easy configuration of class attributes at runtime.

As we can see in Figure 5.22, once we have determined the JVM is the Director, the LoadDirector is instantiated. Using a Preferences node extracted from the configuration file, the LoadDirector obtains information about the particular LoadGenerator and Scheduler it will use. The method loadFromPrefs is inherited from the sminer PrefsBound class and instantiates all preferences attributes contained in the node. This means that, recursively, the LoadGenerator also instantiates and initializes its particular DelayList and InputProcessor. In this particular implementation, the DelayList generates all delays from input. Upon request the LoadGenerator obtains delays, processed them and returns a list of LTTasks. This list is then Scheduled and distributed in as many lists as clients there are. Additionally, the LoadDirector locates the clients and initializes them by setting the exact time when they must start. At the end of its initialization, the LoadDirector is ready to send each schedule to the target client. In this case, we use the SessionInjector to add Session IDs to the delays and create LTSessionTasks that will be distributed using a SessionScheduler.

LEGEND: The Figure below has a color coding for comments and classes, each color corresponds to a different class: light blue for the LoadDirector, red for the Scheduler, pink for the LoadGenerator, dark blue for the InputProcessor and orange for the DelayList.

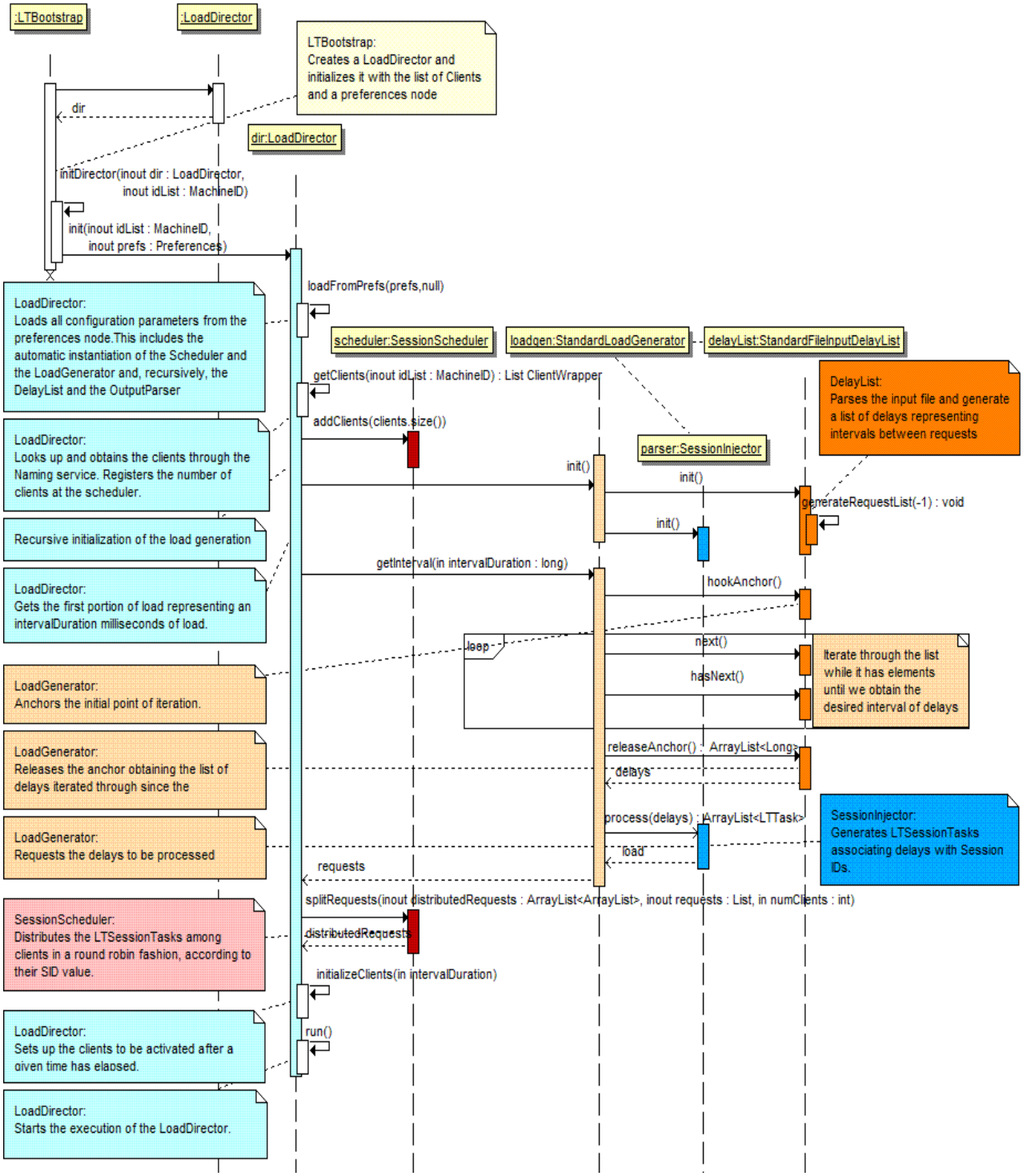


Figure 5.22 : Load Generation and Control Sequence Diagram

2.3.3 Load Execution

We have chosen the TPCWSessionClientWrapper execution to illustrate the management of session requests using EBs with a minimal alteration of the TPCW java implementation behavior.

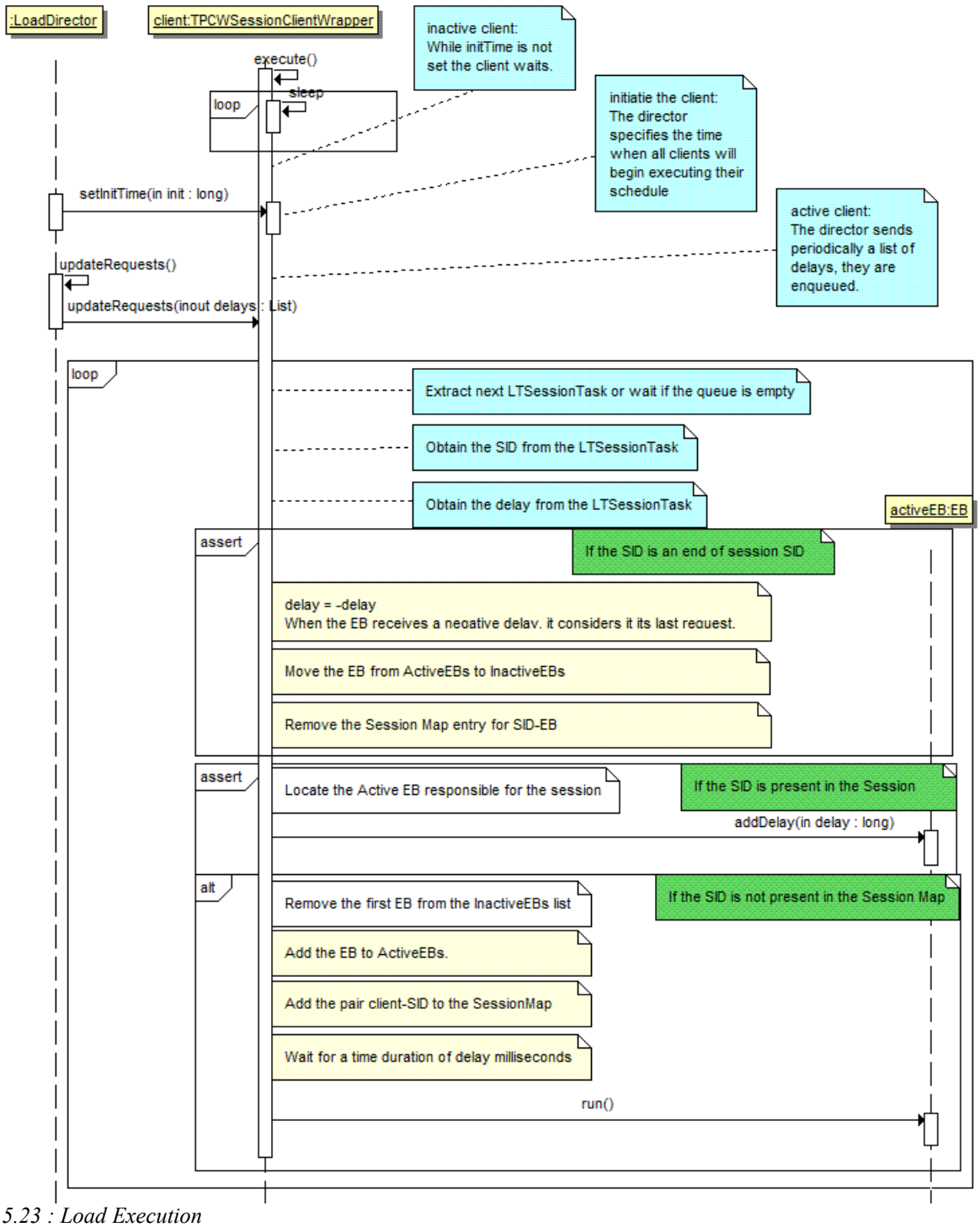


Figure 5.23 : Load Execution

Figure 5.23 presents the events leading to a single request execution using TPCW and sessions. The ClientWrapper presented here has already been initialized. Once it is executing, it waits until it is activated by setting the init time. Once initiated the ClientWrapper polls the request queue for updated requests and extracts the first LTTask, in this case an LTSessionTask. The ClientWrapper determines whether the session is currently being processed by an Emulated Browser or a new one has to be started. After performing the necessary changes the delay contained in the LTSessionTask is sent to the appropriate EB for execution. End of session is managed by identifying negative SIDs, the associated delay is then sent with a negative value. This value is added to the list of EB *think times* between requests. A negative thinktime is interpreted as the last request an EB will send, after it the EB stops running and is recycled.

3 Aspects of the Implementation

In the following lines we will expose several aspects of the implementation we found interesting to mention. The Tester is a 100% Java application which allows it to run on multiple platforms and to be easily extended or modified, we will now present other Java libraries and applications we have used to assist and complement our implementation.

3.1 Sminer Preferences

We have used a utility of the Stream Miner framework developped at the Conrad Zuse Institute in Berlin to allow for easy custom configuration of several key parameters of the Tester, including the components used on a specific execution. This library is similar to the **java.util.prefs** which provides applications with means to store and retrieve user and system preference and configuration data.

Several of our interfaces and implementations are subclasses of both the interface **sminer.util.prefs.PrefsSerialized** and its implementation **sminer.util.dynamic.PrefsBound** which we use to load class attributes from a XMLPreferences node.

PrefsSerialized is a composite class, combining both preferences consumer and producer classes, our interfaces extend PrefsSerialized to allow for all our implementations to be loaded from XMLPreferences nodes.

PrefsBound implements PrefsSerialized and offers methods like "loadFromPrefs" which accepts a XMLPreferences node as an input. This node must be previously created by extracting data from an input file. In short, what Sminer Preferences does is create an object based on an XML node map and then use it to populate classes.

This map is structured as follows: a node represents an instance and contains a map of all the attributes we wish to load from Preferences, additionally it can also contain the name of the particular implementation of this class to instantiate. This feature allows us to configure at runtime which implementation we wish to use. Nodes can also contain other nodes as attributes and are loaded recursively. This hierarchical structure also provides with inheritance of settings, when data is not found under a certain name in a node, it is searched in the upper nodes. The "loadFromPrefs" method allows to manually force an update of all class attributes from a Preferences node.

The Sminer Preferences utilities allow users to setup an execution by editing an xml file, therefore substituting a user interface. Moreover, users can create several configuration files and run the Tester specifying which one to use upon launch.

3.2 TPCW

In the following lines we will expose several changes we have performed to the TPCW java implementation in order to adapt it to the Tester.

3.2.1 Modifications to the EB

The Emulated Browsers used in the TPCW java implementation are started by the RBE and run until they are terminated. Following a Markov Chain they send arbitrary requests to the 14 Servlets the TPCW Benchmark offers. After each request, the EB waits for a duration called thinktime, representing the amount of time a user takes before requesting a new page by following a link. This value is generally calculated using a negative exponential distribution. Once the benchmark is over, all EBs are stopped simultaneously.

We added a thinktime queue to the EB in order to customize the pauses between requests, additionally we have modified the execution method to terminate once the last request after a negative thinktime (indicating end of session) has been sent.

3.2.2 Usage of the RBE

As mentioned earlier, our TPCWClientWrapper implementations contain an instance of the RBE class. In the TPCW java implementation, RBE runs as the main class, using both class and instance methods to configure and run the benchmark. All factories used to instantiate and configure the EBs and other TPCW components require for an instance of RBE. When our ClientWrappers are initialized, they perform the same setup steps as the RBE, using the instance they contain. Putting it simple, the TPCWClientWrapper's initialization method reproduces most of the RBE's main method to setup the test, excluding the actual scheduling and execution of the load.

3.2.3 The ArgumentSource utility

The TPCW benchmark is executed using several configuration parameters passed onto the RBE as arguments such as the number of EBs, the address of the server, the output file, etc.. The arguments are then parsed and added to an argument database for correctness check before several TPCW components -including the RBE- are configured based on them.

In order to provide the TPCWClientWrapper with these values we had the option to make them constant or to allow for custom user input. Choosing the latter option we then designed the ArgumentSource utility, capable to be loaded from a Preferences node which is extracted from a new configuration xml file, exclusive for TPCW. Upon request, the ArgumentSource instance returns all RBE arguments so they can be parsed and stored as if they had been input through command line.

3.3 RMI

We have used the Java Remote Method Invocation libraries to distribute the execution among several JVMs. It has allowed us to setup, locate and use remote services at runtime without programming complexity nor considerable overhead. Moreover, ClientWrapper services can run independent from the Director, which means they can accept schedules from several JVMs and act as task executor services.

On the other hand we had the need to design a remote logging system, and were faced with two options:

The first one was to run the ClientLogger as an RMI and accept requests for log messages from the Director machine. In this scenario the DirectorLogger would poll each client periodically for log data. The issue with this design was scalability, whereas adding one more client to the system would only imply starting a ClientLogger on the client machine, the process of requesting logs would scale linearly at the DirectorLogger. Since the DirectorLogger runs and periodically stores data to disk, stopping to poll for N logs implied a risk to overload its execution time.

We have chosen the second option, clients log periodically at the DirectorLogger, undisturbing its execution. Since a java RMI service accepts concurrent requests, this solution scales better than the first one. On the other hand the problem of consistency of data modified concurrently is solved by using a blocking queue.

In this section we have taken a closer look to the structure of the Tester, starting with its usage and progressively narrowing observations to more specific details of its design and implementation. We have also added several aspects we have found interesting to work on, as well as some discussions about the design choices. Even while writing this section, new ideas and improvements have been brought to consideration, we have exposed an example of extension of the Tester in subsection 1.3.3 and interested readers will find more suggestions at the last section of this document.

Section 6

Conclusions

We conclude this document with an overview of the project planning and costs, several considerations for future work and some personal comments. In the following lines we will summarize and review this experience that represents the transition from student to engineer.

1 Summary

The Tester is a 100% Java tool designed to collect input data representing a schedule of load with the purpose of testing a service. The main features of our design are the ability to collect the input, preprocess it, schedule it in order to distribute it among several JVMs and execute requests against any type of service using a service class programmed by our users.

Other characteristics that define our project are the set of interfaces allowing users to plugin their own versions for every key component of the tool, the Preferences Library for a rich customization of the scenarios, the bootstrap class used for client discovery, and the implementation of a TPCW Benchmark plugin.

The Tester has been built in the scope of a Research Exchange Program of the CoreGrid Network of Excellence, in a collaboration between the UPC and the Zuse Institut Berlin. We have tested its features in the D-Grid Cluster at ZIB, and are currently working on a paper and a technical report to illustrate the results of our research.

2 Project Plan and Costs

In this subsection we present the initial plan we had devised for the project, based on the preliminary requirements, followed by the real plan and the goals we have accomplished. We will also present an evaluation of the estimated cost of the project based on the final plan.

2.1 Initial Plan

As we have mentioned earlier, the project leading to the Tester was originally a much simpler solution to our need for a testing tool. Initially we needed to perform different tests against a service in order to determine its behavior using a real scenario. For this purpose we used load data extracted from server logs.

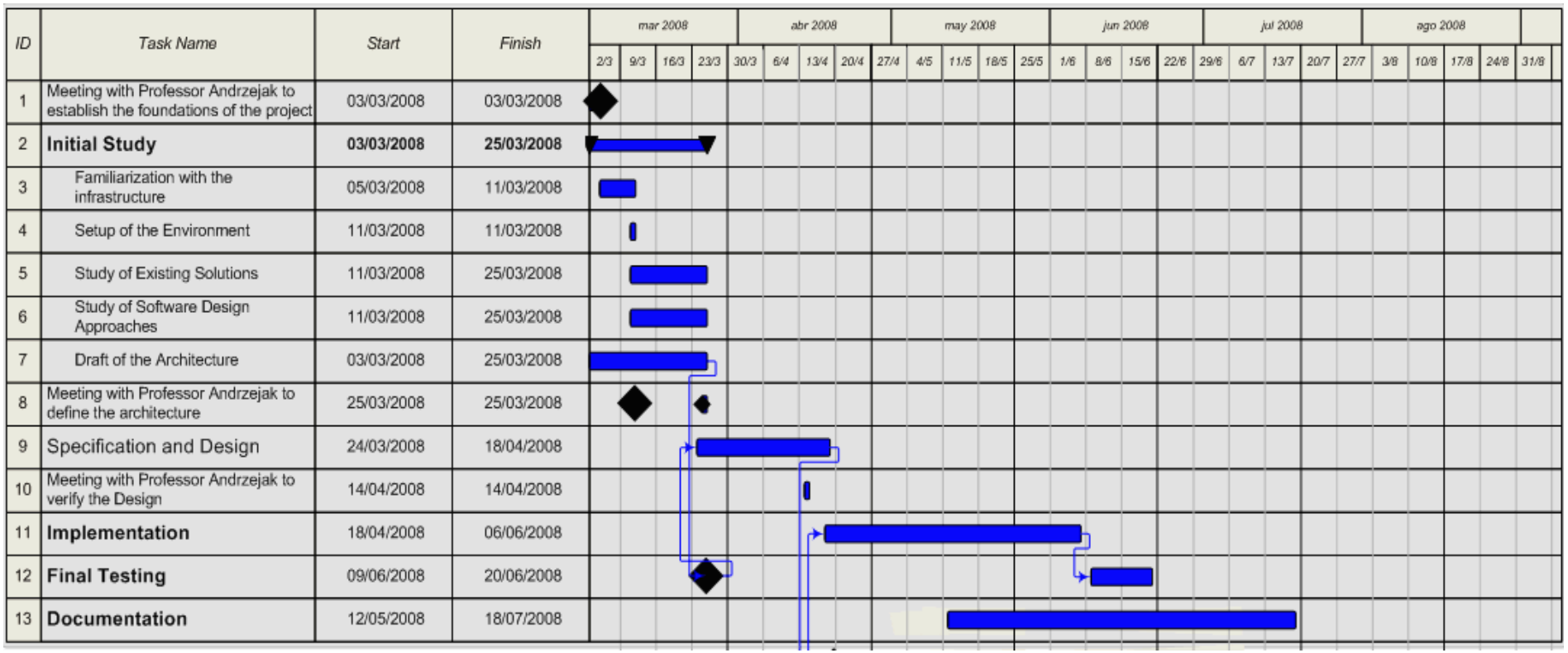
Progressively, due to additional requirements of both critical and optional nature, the Tester has evolved to a more complex architecture with a new purpose: to offer a solution for different users, custom scenarios and all services. Figure 6.1 presents the Gantt Project Plan we had devised at the beginning of our project.

As we can see, we expected to end the implementation on the first week of June, 14 weeks after our arrival to Berlin. As a matter of fact, the Tester was originally meant to be completed at the beginning of April, resulting on a custom small Java application that would fullfill exclusively our needs. The addition of new requirements and ideas, performance complications and other factors led to a more elaborate solution.

Furthermore, as we will expand in the last subsection, complications due to an optimistic scheduling of a project have forced us to not only modify our initial objectives but to continue the project after the publication of this document.

The steps of the initial plan are the following:

- An interview with our host at the ZIB to establish the requirements of the Tester.
- An initial phase of installation, familiarization and study to obtain a certain "know-how" and a draft of the architecture used to discuss different design approaches.
- The specification and design of the project supervised by several meetings with Professor Andrzejak.
- The implementation of our tool
- A testing phase
- The elaboration of this document



ID	Task Name	Start	Finish
1	Meeting with Professor Andrzejak to establish the foundations of the project	03/03/2008	03/03/2008
2	**Initial Study**	**03/03/2008**	**25/03/2008**
3	Familiarization with the infrastructure	05/03/2008	11/03/2008
4	Setup of the Environment	11/03/2008	11/03/2008
5	Study of Existing Solutions	11/03/2008	25/03/2008
6	Study of Software Design Approaches	11/03/2008	25/03/2008
7	Draft of the Architecture	03/03/2008	25/03/2008
8	Meeting with Professor Andrzejak to define the architecture	25/03/2008	25/03/2008
9	Specification and Design	24/03/2008	18/04/2008
10	Meeting with Professor Andrzejak to verify the Design	14/04/2008	14/04/2008
11	**Implementation**	18/04/2008	06/06/2008
12	**Final Testing**	09/06/2008	20/06/2008
13	**Documentation**	12/05/2008	18/07/2008

Figure 4Gantt Diagram: Initial Plan

2.2 Achieved Goals and new Plan

We will now present the new version of the planning of the project and a brief summary of the achieved goals.

Figure 6.2 illustrates the new organisation of our project, as we can see the phases exposed in the initial plan have been divided into multiple subtasks. The design and implementation steps have also been altered to fit the addition of new requirements and ideas as a result of several meetings with our advisors.

The new Gantt Plan is structured as follows:

- An interview with our host at the ZIB to establish the requirements of the Tester.
- An initial phase consisting on :
  - The setup and installation of our working environment inlcluding the setup of a Network File System to emulate a cluster infrastructure.
  - The study of the Sminer Preferences utilities we use for configuation purposes.
  - The study of existing solutions (which we had already overviewed prior to the project).
  - The familiarization with design concepts in Java Programming.
  - A draft of the architecture used to discuss different design approaches.
- After this phase ideas on the architecture were discussed with our supervisors.

According to the initial plan, once we had the structure of our tool we expected to begin a single phase of specification and design and to proceed to the implementation. Nevertheless new requirements and ideas led to a set of iterations, the new Gantt summarizes the three main iterations, which in reality consist of several smaller ones. We now present the real iterations, depicting the different versions of the Tester, the diagram in Figure 6.2 will only represent a combination:

Iteration 1:

- The initial specification and design consisting of use cases and the class structure of a simple scenario where the LoadDirector was responsible for the generation, scheduling and distribution of the load and the ClientWrapper was setup as an RMI executing tasks.
- A phase of implementation, testing, observations, ideas and new requirements

Iteration 2:

- The addition of the LoadGenerator and DelayList interfaces and implementations for load extraction and delay generation
- The addition of several utilities such as the DelayCalculator and the SynchClock
- A phase of implementation, testing, observations, ideas and new requirements

Iteration 3:

- The addition of the Scheduler interface for load distribution and the Bootstrap for client discovery.
- The addition of several utilities such as the InputPattern for custom input and the IPFileNameFilter for discovery of clients.
- A phase of implementation, testing, observations, ideas and new requirements

Iteration 4:

- The addition of the LTTask interface to wrap delays and additional data, the local and remote logging classes and an inputProcessor interface to build LTTasks by adding information to the delays.
- A phase of implementation, testing, observations, ideas and new requirements

Iteration 5:

- The addition of two TPCW Plugins, the LTSessionTask, SessionInjector and SessionScheduler classes.
- The modification of several TPCW Java implementation classes.
- An additional installation of the TPCW Java Implementation framework.
- A phase of implementation, testing, observations, ideas and new requirements

Iteration 6:

- The addition of the Monitor and OutputParser classes to allow for metric extraction and evaluation.
- A phase of implementation, testing, observations, ideas and new requirements

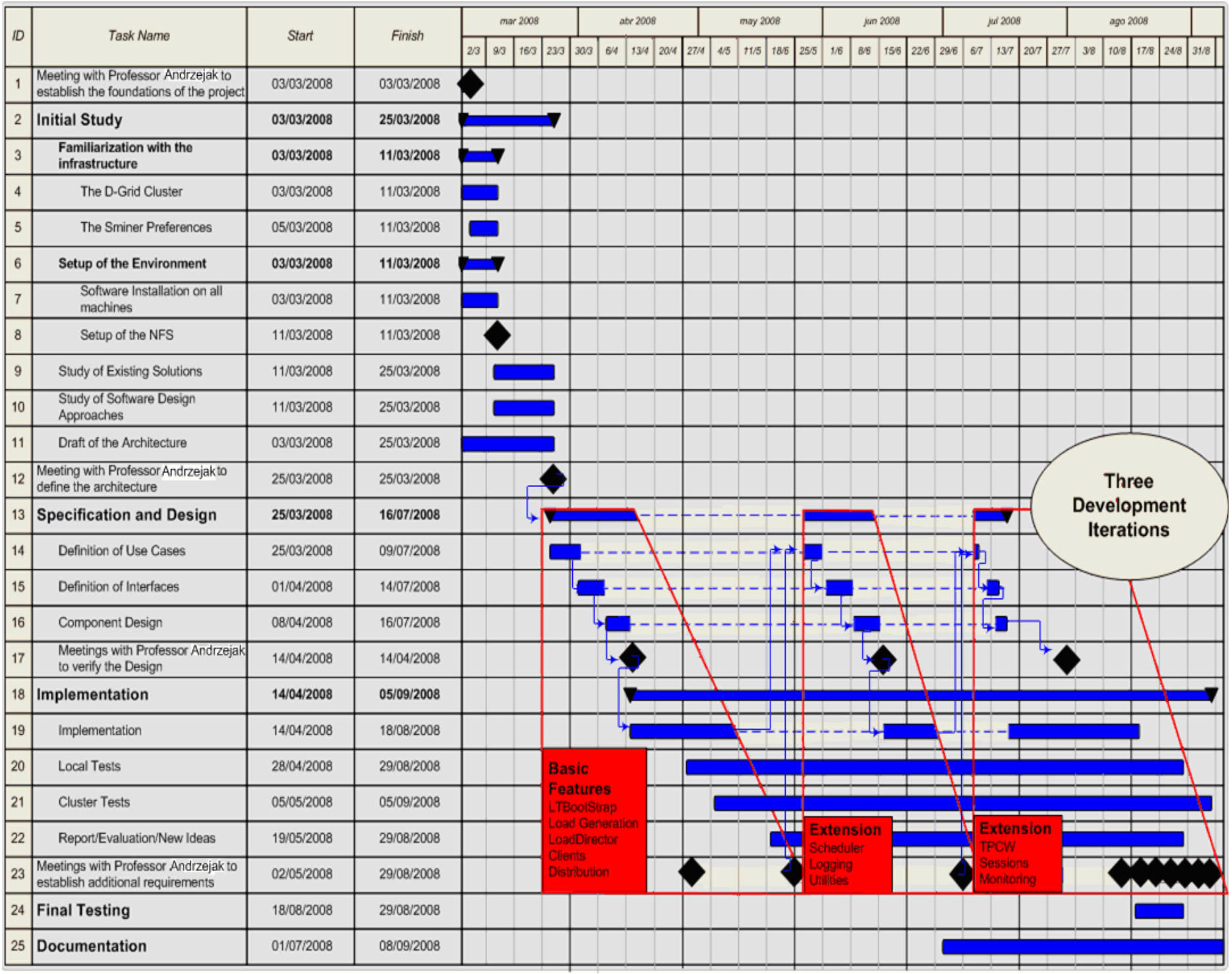


Figure 6.2 : Gantt Diagram: Final Plan

Following the different iterations of the building process, and after establishing a deadline to this project, we performed additional tests and completed the elaboration of this document. Currently we are still modifying several aspects of the Tester, the possibilities it offers, and the different mechanisms of monitoring.

2.3 Costs

This subsection is dedicated to the estimation of the cost of this project. There are two factors that condition and biaise this evaluation :

- This project has been performed as a master thesis therefore it differs from a standard organisation project, it was made by one person and not a team.
- This project has also been financed by a Research Exchange program environment which targets PhD students and experienced researchers therefore its budget differs from a standard one.

The first factor affects the optimal duration of the project, with a team of experienced engineers we estimate the project would have required much less time. The criteria we have established to properly assess the cost of this project are the following:

- The project team consists of an analyst and a developer.
- The salary of the analyst is 35€/hour
- The salary of the developer is 25€/hour
- We have used two server machines:
  - csr-pc23: Intel(R) Pentium(R) IV, CPU 3.06GHz, 3Gb memory
  - csr-pc19: 4x Intel(R) XEON(TM), CPU 2.40GHz, 3Gb memory
- All software we have used (OpenSUSE, OpenOffice, Eclipse) has a free license
- We have also used the D-Grid cluster but the computers in it will not be taken in account.

There is also a second circumstance affecting the number of hours invested in the project. According to the master thesis regulations the project should englobe 750 hours of work, as we can see in the Gantt diagram, due to several factors (such as delays, inexperience, extra hours unaccounted for), the project's time consumption has reached approximately 875 hours.

Extracted from the Gantt diagram in Figure 6.2, the project consisting of 25 weeks can be divided into 11 weeks of Analyst tasks (including preparation, specification, design and documentation, and 14 weeks of Developer tasks (including implementation, testing and documentation). These values translate into a 44% analyst tasks and 56% of development tasks. Therefore we obtain 385 hours for an analyst and 490 hours for a developer.

The result of our assumptions is shown in Figure 6.3, the staff cost for this project taking in account the factors explained above is €. We have also added the material cost relative to 6 months of usage out of a 3 years lease, which means we consider approximately 17% of the total price. With a cost of 680€ in material the total adds up to €.

INFO ROLE	**% of total**	**Total Hours**	**Salary(€/hour)**	**Total**
Analyst	44	385	35	**13.475,00 €**
Developer	56	490	25	**12.250,00 €**
			Staff Cost	**25.725,00 €**
INFO MATERIAL	**Characteristics**	**Cost for a lease of 3 years**	**Time of usage in this project**	**Total**
PC: csr-pc23	Intel(R) Pentium(R) IV, CPU 3.06GHz, 3Gb memory	**1.000,00 €**	6 months	**170,00 €**
Server: csr-pc19	4 processors Intel(R) XEON(TM), CPU 2.40GHz, 3Gb memory	**3.000,00 €**	6 months	**510,00 €**
			Material Cost	**680,00 €**
			Total Cost	**26.405,00 €**

Figure 6.3 : Project Cost

3 Future Work

The Tester is an open project subject to modifications, we have implemented several interfaces to allow for users and developers to tweak and extend our initial implementation. The Preferences library offers the possibility to load a custom implementaiton of an interface and use it as a key component of the scenario. We will now present the ideas that are currently being implemented as well as the ones that would add interesting features to our design.

3.1 Things we are working on

We are currently working on a research project which aims to determine the behavior of a service under different circumstances. More specifically we intend to put a service under real load while injecting memory leaks and load bursts. For this purpose we have already implemented a servlet that runs in a Tomcat server whose purpose is to vary the memory consumption emulating memory leaks. This feature is complemented by a LeakDirector and a LeakClient to allow for scheduled, organized and distributed leak injection. Additionally we are working on improving the monitoring class to be capable of examining the service response and deciding whether we should inject load bursts, drop a client, or change the scale of the load.

Using these new features of the Tester we will be capable of observing the service's behavior to determine its limits and to what extent memory leaks affect its performance. We are currently working on a publication in association with Professor Artur Andrzejak at the ZIB.

3.2 Ideas

During the process of building the Tester several new ideas and innovations have appeared. We now offer a list of some of them we found interesting to comment:

- The adaptation of a Web based interface to the LoadDirector and the Monitor in the following manner:
  - Adding RMI access to both classes.
  - Creating a collection of servlets to interact with the RMIs with the following features:
    - Allowing a LoadDirector and Monitor to register at the servlet
    - Allowing clients to interact with the LoadDirector and the Monitor through the servlets

- The creation of an implementation of the DelayList, the InputProcessor and the ClientWrapper to allow for multiple possibilities of building LTTasks.

- Adding a new feature modifying the LoadDirector, the Scheduler and the Monitor that customizes the distribution of the load depending on the current load and performance of the client machines.

- Establishing an Exception and Error handling system with a structure of advices for optimal usage.

- Building additional classes to allow for a fast creation of service clients similar to the ones we found on the tools we presented in the related work section.

4 Personal Comments

I finally conclude this document with a few personal comments about the experience of both building a software for my master thesis and doing so in a research exchange program.

I have had the opportunity of making this project in an international environment at the ZIB, an institute where researchers from several countries in Europe are involved in different projects. I have assisted to several presentations on the subject of different state of the art research projects such as the XtreemOS, and MediGrid projects.

The knowledge I acquired during my Computer Science studies at the FIB has been complemented by learning from the experience of various researchers and PhD students with different educational origins. I have also been able to assess my expertise and to have a clear idea of what capabilities I have developed during the course of my studies. A gratifying output was to find myself discussing evenly with my colleagues at the ZIB about several subjects, even being of help in some cases. I progressively discovered I had a solid base knowledge and know-how that I hadn't necessarily noticed before since my professional interactions had mostly been with other FIB students. This experience has therefore allowed me to build a more clear image of what I am capable of, and to value my work.

During the realization of this project I was also coursing two optional subjects at the FIB and the coincidence made it so that both have helped me a lot in my work. HITI (Information Abilities for Information Technologies) has assisted me in the organisation of the master thesis while SODX

(Distributed and Network Operating Systems) was constantly complementing my work and also allowed me to understand more easily some of the GRID projects that are being realized at the ZIB. For example, one of the tasks of SODX is to review several research papers as a practice, on the other hand at the ZIB I was handed real papers for review, therefore both activities complemented eachother.

During the process of designing the Tester, I had the need to learn several concepts and methodologies in order to reach the established goals. I extended my knowledge on the functionment of Tomcat, the possibilities of Java, distributing tasks, refactoring, and many other factors affecting the realisation of this master thesis. On the other hand I also acquired know-how and learned about less technical aspects such as communication: it is necessary to establish a common ground and language to be able to work efficiently, planning: an initial plan will always be too optimistic and should be reconsidered, organization: the amount of work to do can be doubled if it is not properly organized and documented (e.g. After a couple weekends I learned to leave notes on my desk on Friday describing what I had been doing and what I had to do next in order to remember on Monday), self confidence: it is important to value the potential one's ideas can have, or at least give them a chance and mention them.

Overall the experience was enriching and a process necessary to mature the knowledge acquired in a concealed academic environment. I also consider important the fact that in many circumstances there were issues I had to face on my own, and several decisions I had to base on my criteria -despite the valuable assistance and feedback from my supervisors- in opposition to the student community and teacher support I was used to. I also had the chance to work in contact with very interesting people, to learn about different work habits and methodologies, and specially to see there is no clear line drawn after one finishes their studies that separates engineers and students because even PhD students find themselves constantly needing to learn and having to tell themselves what they are capable of. All in all, there is much I have gained in this deal, despite the stress and lack of sleep, and I conclude this master thesis with a broad smile and a warm feeling of satisfaction.

Annex 1

Bibliography

1 Books

Q. H. Mahmoud. *Distributed Java Programming with JAVA*, *Manning Edition 1999.* ISBN:1884777651

A. Shalloway, J. R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Addison Wesley Professional 2001. ISBN-10: 0-201-71594-5

2 Articles

INTRODUCTION AND INITIAL STUDY

Zuse Institute Berlin. www.zib.de

E. Marilly, O. Martinot, S. Betgé-Brezetz, G. Delègue. *Requirements for Service Level Agreement Management.* IP Operations and Management, 2002 IEEE Workshop, pp. 57 - 62

Adam Grummitt. *The Top 10 Myths of Performance Analysis and Capacity Planning(2003).* White Paper, Metron Technology, Ltd Metron-Athene Inc (2003).
available at: http://www.metron.co.uk/documents/whitepapers/top_ten_myths_web.pdf

T. A. Bass. *The Wired Interview to Nicholas Negroponte.* Bookshelves Archive [Website] http://archives.obs-us.com/
available at: http://archives.obs-us.com/obs/english/books/nn/bd1101bn.htm

R. Spiegel. *When Did The Internet Become Mainstream?(1999).*
EcommerceTimes [Website] www.ecommercetimes.com
available at: http://www.ecommercetimes.com/story/1731.html

Standard Performance Evaluation Corporation. SPECweb99 Benchmark.
[Website] http://www.spec.org

S. Manley, M. Seltzer, M. Courage. *A Self-Scaling and Self-Configuring Benchmark for Web Servers.* Joint International Conference on Measurement and Modeling of Computer Systems. Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems. ISBN:0-89791-982-3, pp. 270 - 271.

B. Hyun Lim, J. Ryong Kim, K. Hyun Shim. *A Load Testing Architecture for Networked Virtual Environment.* Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference. ISBN: 89-5519-129-4,vol 1.

D. A. Menascé. *Load Testing of Web Sites*. IEEE Internet Computing. ISSN:1089-7801 2002,vol 6 , Issue 4, pp 70 – 74.

G. Bracha. *Generics in the Java Programming Language (2004).*
available at: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

I. Hussain. *Centralized Logging with JDK 1.4.*
available at: *www.imranweb.com/RemoteLogging.pdf*

HTTPERF

httperf homepage. http://www.hpl.hp.com/research/linux/httperf/

D. Mosberger T. Jin. *httperf -- A Tool for Measuring Web Server Performance(1998).*
available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.4800

T. Bullock. *httperf Web Workload Generator Quickstart Guide.* WWW2007 Web Tools Developer Track.
available at: http://www.comlore.com/httperf/httperf-quickstart-guide.pdf

JMETER

Jmeter homepage: http://jakarta.apache.org/jmeter/

Jmeter user manual: http://jakarta.apache.org/jmeter/usermanual/index.html

J. Buret, N. Droze. *An Overview of Load Testing Tools (2003).*
Available at: http://clif.objectweb.org/load_tools_overview.pdf

LOADRUNNER

LoadRunner homepage: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__
LoadRunner tutorials: http://motevich.blogspot.com/2008/01/loadrunner-tutorials.html
LoadRunner architecture: http://www.wilsonmar.com/1loadrun.htm ç

THE GRINDER

The Grinder homepage: http://grinder.sourceforge.net/

D. George. *Stress Testing with The Grinder and Cactus.*
available at: http://www.abcseo.com/papers/grinder.htm

Travis Bear. *Shootout: Load Runner vs The Grinder vs Apache Jmeter.*
available at: http://blackanvil.blogspot.com/2006/06/shootout-load-runner-vs-grinder-vs.html

PUSHTOTEST

PushToTest homepage: http://www.pushtotest.com/

PushToTest tutorials: http://docs.pushtotest.com/docs/tutorial.html

CLIF

Clif homepage: http://clif.objectweb.org/

B. Dillenseger, E. Cecchet. *CLIF is a Load Injection Framework.* OOPSLA 2003 Workshop October 26, 2003. Anaheim, California, USA.
available at: http://dsrg.mff.cuni.cz/projects/corba/oopsla-workshop-03/Dillenseger-Cecchet.pdf

B. Dillenseger. *Using CLIF load injection framework for performance evaluation – a use case with Speedo JDO implementation.* Middleware 2005, ACM/IFIP/USENIX 6th International Middleware Conference [Website] http://middleware05.objectweb.org/
available at: *http://middleware05.objectweb.org/WSProceedings/demos/d2_Dillenseger.pdf*

TPCW

J. Kiefer. *TPC-W Java Implementation.*
available at: http://mitglied.lycos.de/jankiefer/tpcw
see also: http://www.ece.wisc.edu/~pharm/

Z. Hong, F. Xin, L. Qiu Hui, K. Lü. *The Design and Implementation of a Performance Evaluation Tool with TPC-W Benchmark.* CIT. Journal of computing and information technology ISSN 1330-1136 2006, vol.14, n^{o}2, pp. 149-159.
available at: http://cit.zesoi.fer.hr/downloadPaper.php?paper=713

L. Zhu, I. Gorton, Y. Liu, N. Bao Bui. *Model Driven Benchmark Generation for Web Services.* International Conference on Software Engineering. Proceedings of the 2006 international workshop on Service-oriented software engineering ISBN:1-59593-398-0, pp 33-39.

TPC BENCHMARK™ W, (Web Commerce) Specification, Version 1.8. The TPC Council
available at: http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf

W. D. Smith. *TPC-W: Benchmarking An Ecommerce Solution.*
available at: http://www.tpc.org/tpcw/TPC-W_Wh.pdf

Q. Zhang, L. Cherkasova, E. Smirni. *A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications.* International Conference on Autonomic Computing Proceedings of the Fourth International Conference on Autonomic Computing 2007, ISBN:0-7695-2779-5 p27.
available at: http://www.hpl.hp.com/personal/Lucy_Cherkasova/papers/TPCW-ICAC.pdf

D. A. Menascé, *TPC-W: A Benchmark for E-commerce.* IEEE Internet Computing, vol.6, no. 3, May/June 2002, pp.83-87.

RMI

B. Vandegriend. *Understanding Java RMI: A Simple Tutorial (2006).*
Basil Vandegriend:Professional Software Development [Website] www.basilv.com
available at: http://www.basilv.com/psd/blog/2006/java-rmi-tutorial

K. Qureshi, H. Rashid. *A Performance Evaluation of RPC, Java RMI, MPI and PVM.* Malaysian Journal of Computer Science, 18 (2). pp.38-44. ISSN 0127-9084
available at: http://mjcs.fsktm.um.edu.my/document.aspx?FileName=339.pdf

RMI Testing Ideas.
available at: https://svn.apache.org/repos/asf/harmony/enhanced/classlib/archive/modules/rmi2/doc/testing/Testing%20RMI%20Version%201.0.pdf

FUTURE WORK

M. E. Crovellay, M. Harchol-Balter, C. D. Murtaz. *Task Assignment in a Distributed System: Improving Performance by Unbalancing Load (1998).*

K, Ishibashi, T. Kanazawa, M. Aida. *Active/Passive Combination-type Performance Measurement Method Using Change-of-measure Framework. Global Telecommunications Conference, 2002. GLOBECOM apos;02. IEEE Volume 3, Issue , 17-21 Nov. 2002 Page(s): 2538 - 2542 vol.3*