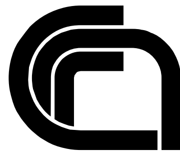
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PUblication MAnagement



Consiglio Nazionale delle Ricerche

Design and implementation of an EPP load generator

Marina Buzzi, Marco Conti, Enrico Gregori, Giuseppe Valente

IIT TR-02/2005

Technical report

Febbraio 2005

Istituto di Informatica e Telematica

Abstract
The RFC3730 describes the Extensible Provisioning Protocol (EPP), an XML-based protocol created for providing a standard Internet domain name registration protocol on behalf of domain name registrars and registries.
The goal of our study is to understand the key elements in performance of an EPP system designed to automate the domain name registration process. For accomplishing this task, as first step, we designed and implemented an EPP Load Generator which creates synthetic traffic of domain name registrations.
In this report the architecture of the EPP load generator is discussed in detail.

Introduction

The main goal of our study is to understand the key elements in performance of an EPP (Extensible Provisioning Protocol) system designed to provide an automated domain registration process. Generally speaking, the term benchmarking refers on running a set of representative programs on different computers and networks and measuring the results. Time and rate are the basic measures of system performance. From the user's viewpoint, the execution time is the best indicator of system performance. From the system's manager viewpoint is relevant the number of transactions the server is able to manage per minute [3]. In our study, to be able to understand the server's capacity, we create a controlled test-bed environment which allows us to analyze EPP server performance in a simple and effective way. We are mainly interested in observing Quality of Service perceived by ISPs (average response time) and carrying out Server Tuning, with particular focus on discovering bottlenecks.
The registration of domain names involves numerous SW components for user authentication, accounting, transaction, data storage, backups, and thus the overall architecture is quite complex. In this kind of scenario, system simulation (EPP server, application server and database) is very difficult and, if not adequately designed, may furnish unrealistic results. An alternative is to measure live systems (Fig.1). However this approach of directly evaluating the performance of a server suffers from difficulties related to the highly unpredictable behavior of the Internet and the need for non-intrusive measurement of a live system.

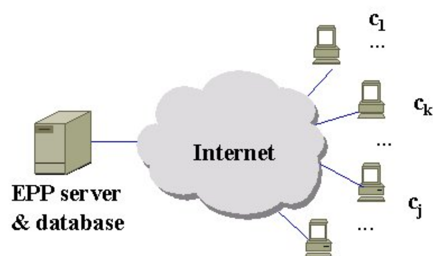


Fig. 1: an EPP server in the real world

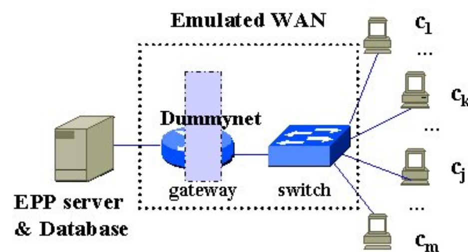


Fig. 2: EPP server in a WAN emulated environment

A balance between these two approaches is possible by evaluating the server through generation of synthetic EPP traffic in a controlled environment, as shown in Fig. 2. Specifically, the EPP traffic is generated by load generators which run on the $c_1..c_m$ clients while the WAN behavior is emulated introducing network delays by means of SW (i.e. the dummynet option furnished by BSD OS).
Several studies have been carried out in web server performance, but in our knowledge not any in EPP performance. However since in our scenario the EPP protocol relays on TCP connections, some considerations for HTTP traffic generation can be analogously done for EPP load generation. Generating heavy and realistic traffic with a small number of client machines is difficult. A simple load-generating scheme, used in past studies [4], [6] that equates client load with the number of client processes/threads in the test systems (adding client processes to increase the total

client request rate) is limited by some characteristics of the TCP protocol [5]. Thus, to generate a significant rate of requests beyond the capacity of the server, using this simple scheme, one must employ a huge number of client processes. This simple approach does not work well; in fact when generating synthetic HTTP requests, care must be taken that resource constraints on the clients do not accidentally distort the measured server performance. The primary factor in preventing client bottlenecks from affecting server performance results is to limit the number of simulated clients per client machine. In addition, it is important to avoid I/O operations in the simulated clients, as discussed later.
In the following sections we describe the first phase of our research. We first illustrate basics of the EPP protocol and then discuss the architecture of an EPP load generator.

The Extensible Provisioning Protocol

The RFC3730: the Extensible Provisioning Protocol (EPP) has been published by the IETF Network Working Group on March 2004. EPP is a client-server XML-based protocol designed for creation and management of objects stored in a shared central repository. Although the protocol was originally created for providing a standard Internet domain name registration protocol on behalf of domain name registrars and registries, it is generic and should be applied in different fields. Specifically, this protocol furnishes a mean of interaction between a registrar's applications and registry applications [2].
EPP v. 1.0 provides four basic service elements: service discovery, commands, responses, and an extension framework.
Basically an EPP client sends a command to the EPP server and receives a response from the server. It is important to notice that EPP commands are atomic and idempotent, i.e. multiple execution of the same command have the same effect on system state as executing the command only once. The EPP server processes commands in the order they are received from an EPP client, since to preserve the temporal order of client command arrival is fundamental to correctly solve collisions in domain name registration. Commands and response are XML messages, with the MIME type: application/epp+xml.
EPP commands are split in three categories:

- Session Management Commands allow clients to establish and close persistent sessions with an EPP server:
    - <login>
    - <logout>
- Query Commands carry out read-only operations for retrieval of object information:
    - <check>
    - <info>
    - <poll>
    - <transfer>
- Transform Commands perform read-write operations for object creation and management. If the transform command requires an offline review, the server acknowledges that the requested action is pending and notifies (the client) when offline processing of the action has been actually completed.
    - <create>
    - <delete>

- <renew>
- <transfer>
- <update>

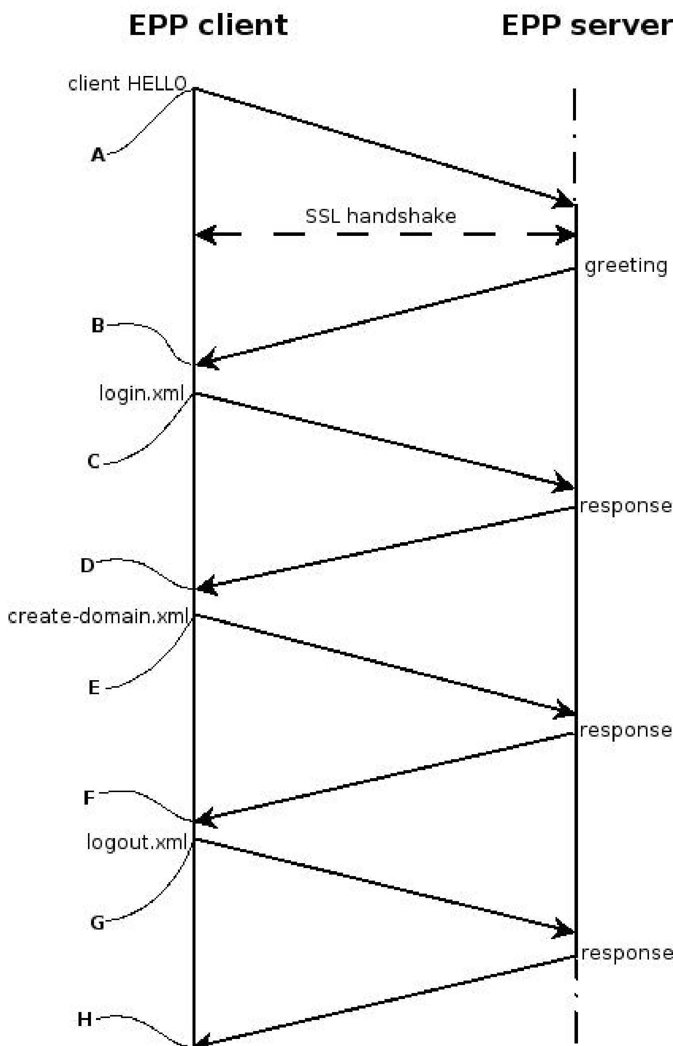
Since the aim of this work is to carry out a study on a EPP architecture in order to test server performance and discover bottlenecks under the hypothesis of a very strong competition between ISPs for domain name registrations, we focus our measurements on generating a high load of the <create> command in order to bring the EPP server beyond its capacity and observe the response time.
A typical single domain name registration operation requires following client-server interactions:

- C: <hello>
- S: <greetings>
- C: <login>
- S: <login> response
- C: <create>
- S: <create> response
- C: <logout>
- S: <logout> response

Each EPP message exchanged between client and server must be a well formed XML file starting with the <epp> tag and ending with the </epp> tag:

```
<epp xmlns="urn:ietf:params:xml:ns:epp-1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:ietf:params:xml:ns:epp-1.0
        epp-1.0.xsd">
</epp>
```

EPP load generator



Our analysis focuses on the most important operation that a registrar would perform: the creation of a domain name. This operation is very delicate, since we have to figure out if a client in any location (and with any bandwidth available, in a specific range) is some way disadvantaged in the competition with other clients for domain name registration.
The first thing we all agreed on as we approached the code was to introduce something to record the times, during interaction with the EPP server. Our first hypothesis was to introduce two timing indexes, but to explain this we need to take a close look at the interaction between EPP client and server. Fig. 3 shows the basic interaction between client and server when a domain is created. We named commands as {command}.xml, where {command}.xml is a string obtained by

prepending the length of the so called “EppString” that represents the command to the command itself, as required by the EPP protocol. We omitted the SSL handshake since it is managed atomically by the Java implementation and its details are not relevant for our study. We used letters to indicate the instants measurable by the client, as those constitute all the timing values noticeable on the client’s side. Obviously, our first thought was to measure the total duration of the connection, which is from A to H. In addition, it would surely be significant to break down that time into all the intervals indicated on the figure, to discover any bottleneck, but we preferred to start measuring the overall connection time to search for bottlenecks as we will explain when talking about the implementation.

Architecture

The first phase of our work simply concerned extending the capabilities of the EPP client example furnished by our DNS Belgium partners to implement the load generator, so we just introduced a connection time value to be logged from each client. We would connect each client separately with a cycle in a bash script. This was only the first attempt and we knew well that this was not going to work, because after a few clients (71 on a 1.07 GHz PowerPC G4) we ran out of memory and we were unable to stop them.

The first thought was to move to multithreading, to raise the number of clients loadable by a single host, reduce the number of hosts to distribute the clients on and so probably reach the server’s “saturation” earlier. The load generator we built was intended to carry out the activity shown in Fig. 4. It was clear that for the software structure needed to realize a good piece of code satisfying these requisites, a class was no longer enough. Then, we decided to design a package that would implement everything needed for our measurements, but also for the post-processing of the values obtained during them. This resulted in the five classes shown in Fig. 5, grouped together in a package we called EppLoadGenerator:

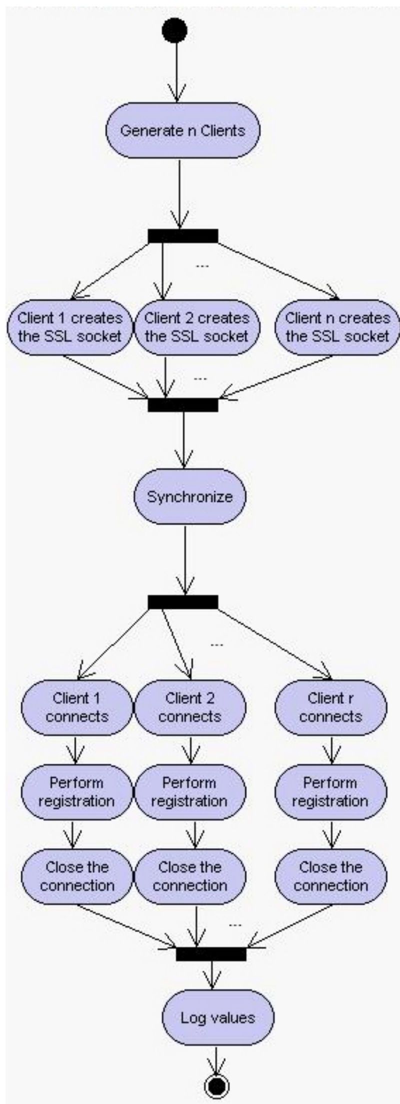


Fig. 4: load generation activity

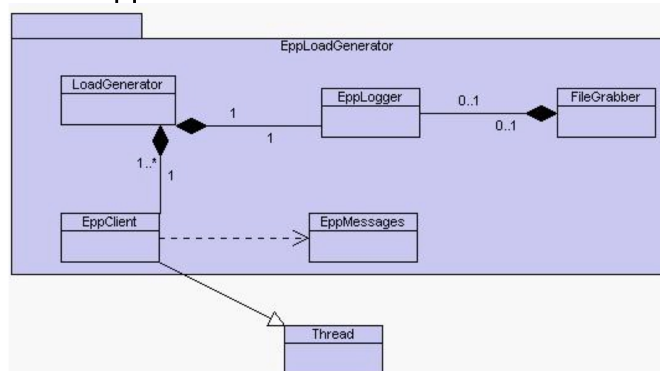


Fig. 5: EppLoadGenerator package classdiagram

- *LoadGenerator* is the main class, which coordinates all the generator’s activity
- *EppClient* is basically the same client implemented from our DNS Belgium partners, but automated to perform a domain creation, return the time values we’re interested in and die

- in *EppMessages* we separated the “EPP aware” side of the client to enhance it somewhat
- *EppLogger* manages the output of the generator directly from the client or with previous computations if desired
- *FileGrabber* is designed to manage EppLoadGenerator log files for further post-processing

We will now provide more details about each single class (except for the FileGrabber, whose use will be clear when we will talk about distributing load generators) to explain their features and the choices we had to make during the implementation phase.

EppClient

The Java implementation we started from modeled a rough EPP client, that performed basic operations like login and logout, but was also able to read xml formatted files and send them as other commands. Everything necessary for the connection was provided: a client certificate and a method to perform the SSL handshake, as well as an account on DNS Belgium’s tryout server with which we were able to create the contact and billing info needed to register domains and perform any other operation. The structure of the client was perfect to start working on the load generator because of its simplicity and modularity: any single operation was performed as a self explicative method within the class EppClient. All we did about this class was to extract the part concerning EPP messaging (as we will see in EppMessages), insert the timestamps measurements we need, and make the client perform the operations shown in Fig. 3 (and at the bottom of Fig. 4) automatically, without any interaction with the user.

LoadGenerator

The LoadGenerator class is the core of the package. It basically creates an array of EppClient classes, created as threads. This fact made the clients naturally prone to synchronization: we could start a certain number of clients/threads and then put them on wait just before the connect method. Having them all interrupted made it easy (a while condition was enough) to unlock them with a given error, that we set as 1000 milliseconds because we want to look at a certain number of connections per second. This is part of what we mean with the synchronization activity in: this activity will be clarified when discussing about distributing load generators, since there are actually two levels of synchronization in our implementation of the load generator. Here we only explain is the second level, which aims at thread level synchronization.

EppMessages

The EppMessages class provides some control on connection behavior: this point is essential because if for example the creation of a domain fails, the data collected from the test would not be consistent because the measured time does not include the time needed to write on the database (and do the necessary lock on the resource on the server’s side). Considering possible bottlenecks, we also thought about the disk on the client (or at this point we should better say the generator), so putting these thoughts together we implemented this class, which is composed of static fields and methods that solve many problems. Specifically the EppMessages class carries out following functions:

- groups the xml commands together, so that they are loaded in memory from the Java VM and reading them from files on the local disk is no longer necessary

- associates an integer value to each command, so that command selection becomes cheaper than it would be if it was realized as a comparison between strings
- sets the domain name as iit{timestamp}.be, so that the create operation can be fully automated
- provides a minimal interpretation of the server responses, by parsing the EPP strings received from the server and checking the code in the <result> field of the <response>. If the code is not the one expected in the successful case, it returns the content of the <msg> field of the <result>, together with the analogous <dnsbe:msg> field of the <extension> part of the response for easier interpretation.

EppLogger

The EppLogger class was initially designed to log the output. Since we did not know a priori which values would be significant, we thought that the best thing to do was to create a tool as scalable as possible. The EppLogger is basically a matrix of times (except for the last value in each row, which represents an index) with a set of methods to operate on it and to extract the values and the statistics that we want to know. The rows of the matrix represent the clients, the columns represent the values to log (the last value of each client is not really a value to log but an index representing the order of termination of the clients); the LoadGenerator can pilot this matrix as we wish: it can create it and fill it with the whole values shown in Fig. 5 as well as it can just fill it with the total connection times of each client.

This brings us to a choice we had to make: we had to find a trade-off between logging everything, and making the clients run fast. If we make each client log the 8 values shown in Fig. 3, then we would have to lock the times' matrix 8 times for each client: this would surely be a bottleneck (and we tested it) and it would be a bottleneck during the connection, which means that the increase of time produced would be included in the times logged and we do not want that. The alternative would be to keep a variable for each time value on each client, but there is no need to explain that this would be a cost too, since we are probably going to generate a very high number of clients. We chose to keep the implementation very light, but at the same time be prepared for anything: we kept the matrix architecture making each client fill it with the connection time after the connection was closed, so the mutual exclusion of the operation would not influence any time logged. Introducing more time variables in the client is rapid to do, and the times matrix is still scalable enough to be managed as we wish.

The active operation of the logger occurs after clients terminate: the LoadGenerator tells it to manage the values we want to log. Specifically it:

- orders them by completion instant of the client;
- logs the values;
- logs the instant when the values were recorded;
- appends the mean to the log;
- appends the 90th, the 95th and the 99th percentile to the log;

The first and the third operation are very important.

The first operation is needed to measure the slow start we were talking about: once this slow start has been measured, we can avoid logging the transient phase by using the constructor overloading provided within the EppLogger. With it is possible to construct a logger specifying the transient phase length measured in number of clients.
The third operation introduces us to another part of the implementation which is fundamental for testing: distributing load generators.

Distributing load generators

From the beginning, we intended to run the load generator distributed on several hosts. The synchronization between those hosts would have been easily solved by using crond to launch the generators, but the fact that we introduced a second level of synchronization between the threads, as well as the time required to load the Java VM on each host, made it necessary to find an additional solution to solve the problem. The diagram shown in Fig. 6 expands the synchronization activity in Fig. 4. The "setup time" shown is that first level of synchronization we mentioned in the previous paragraph: it is an interval calculated from the time the load generator starts (which will have to be the same for all the generators) by looking at each host's clock. If the clocks of all hosts are synchronized among themselves, we can assume it to be an absolute time.

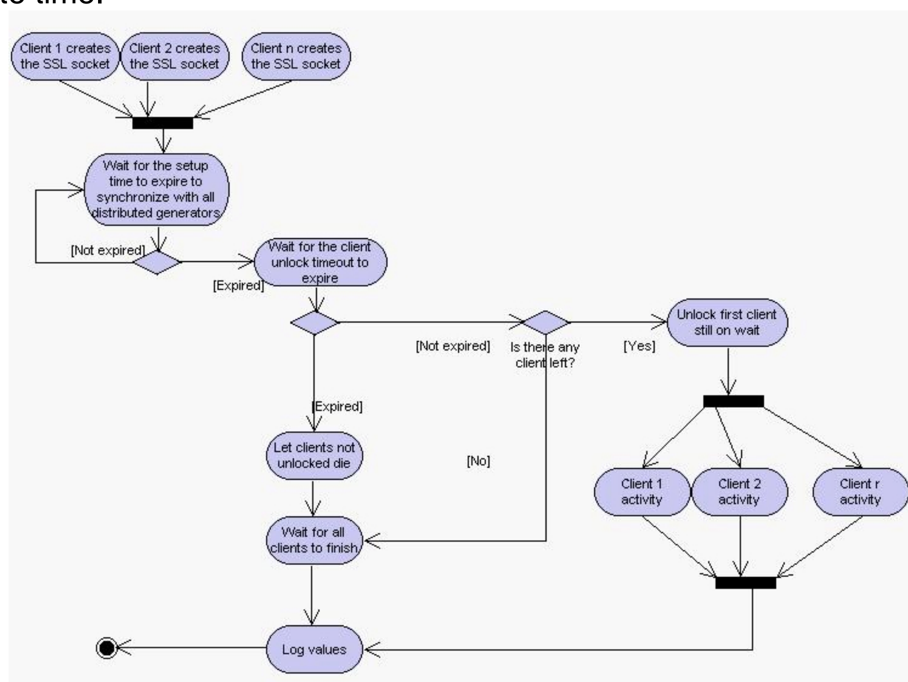


Fig. 6: synchronization activity of the load generator

FileGrabber

Once we solved the problem of running load generators in parallel, it still remained to log all the activity of those load generators together. That is why we made the logger save the instant when each value was recorded: with these values we can rebuild the time sequence in which things occurred on different hosts. To provide logs that are comprehensive of what happened on all client hosts, we implemented a utility we called FileGrabber. FileGrabber is a class in the same package of LoadGenerator that basically reads the files and creates an EppLogger whose times' matrix contains all the values collected from the files. We had to enhance the logger to reorder those values using the timestamps that each client logged, and the rest of the operation was pretty much the same performed from the logger in its normal activity on a single host.

As stated above this FileGrabber is a utility since it collects everything that was left over and that we had to implement in a later phase. It makes distributed logs transparent to the user, but it also provides global statistical knowledge, such as confidence intervals on the values logged in the files. This last option does not even use the logger, and thus acts like a standalone utility. This use of the package clarifies why we logged mean values of each generator by appending them to a file. Fig. 7 and Fig. 8 show respectively a class diagram which summarizes the structure of the entire package and the whole load generation activity (still keeping a high level in the description).

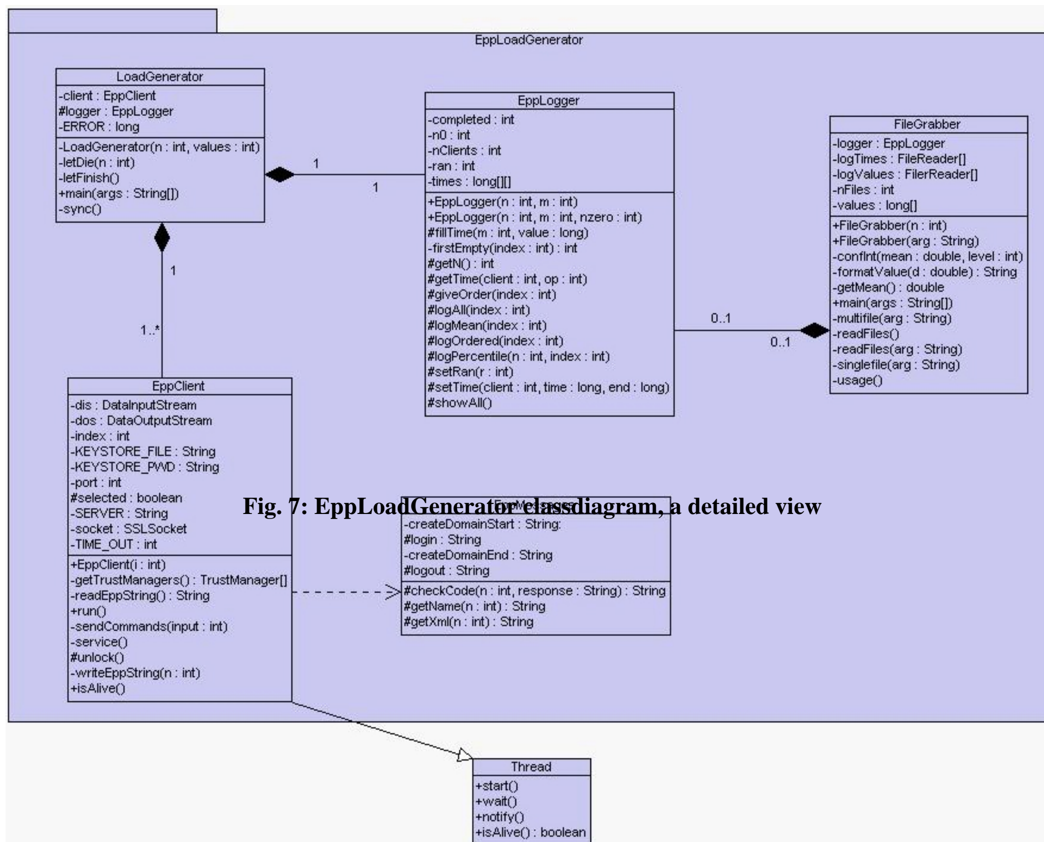


Fig. 7: EppLoadGenerator classdiagram, a detailed view

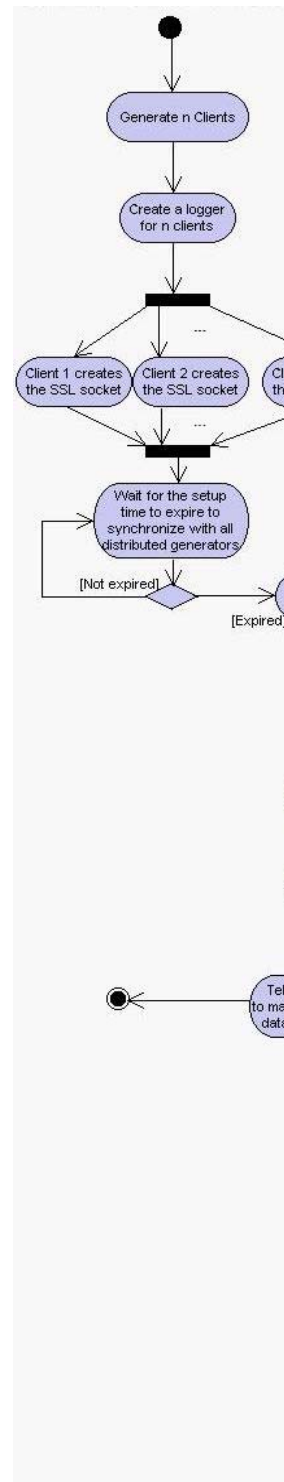


Fig. 8: overall activity diagram

References

1. Registro ccTLD .IT- http://www.nic.it/

2. S. Hollenbeck. RFC 3730, March 2004, http://www.ietf.org/rfc/rfc3730.txt
3. Daniel A Menascé, Virgilio A. F. Almeida. Capacity Planning for Web Services. Prentice Hall.
4. ACME Laboratories Freeware Library, THTTPD Web Server, http://www.acme.com/software/thttpd.
5. Banga G., Druschel P., Measuring the Capacity of a Web Server Under Realistic Loads, World Wide Web Volume 2 Issue 1-2. Kluwer, Dordrecht (1999) 69-83.
6. The Standard Performance Evaluation Corporation (SPEC), SPECWeb99 Benchmark, http://www.spec.org/web99/.