Distributed Load Tests with TTCN-3

George Din[1], Sorin Tolea[1], and Ina Schieferdecker[1,2]

[1] Fraunhofer FOKUS, MOTION, Kaiserin-Augusta-Allee 31,
10589 Berlin, Germany
[2] Technical University Berlin, Faculty IV,
Straße des 17. Juni 135, 10623 Berlin, Germany
{din, tolea, schieferdecker}@fokus.fraunhofer.de

Abstract. The design of TTCN-3 focused on extensions to address testing needs of modern telecom and datacom technologies and widen the applicability to many kinds of tests including performance tests. One of the most important features of TTCN-3 is the platform independence which allows testers to concentrate on the test specification while the complexity of the underlying platform (i.e., operating system, hardware configuration, etc.) is left behind the scene. As far as the test distribution is concerned, TTCN-3 provides the necessary language elements for distributed tests. This is however supported in a transparent fashion so that the same test may run either locally or distributed. The distributed execution of a test enables the execution of test components belonging to one test configuration on different computers (the test nodes), sharing thus a bigger amount of computational resources. Test distribution is a research challenge when it comes to the problem of how to distribute the test components efficiently on the test nodes. Specifically for load testing – a particular kind of performance test – we investigate strategies to distribute tests on heterogeneous hardware in order to use the hardware resources of the test nodes efficiently.

1 Introduction

Performance testing is a qualitative and quantitative evaluation of a System under Test (SUT) under realistic conditions to identify problems for scalability or usability aspects under heavy load and to collect measurements as success/fail rate, response times or round-trip delay. Although performance testing is often used in different ways, performance testing usually determines how fast a system reacts or how much load a system can handle. The literature distinguishes [13]: load, robustness, stress, or volume testing. *Load testing* simulates various loads and activities that a system is expected to encounter during production time. The typical outcome of a load test is the level of the load the system can handle but also measurements like fail rate, delays under load etc. Load testing helps detecting problems of the SUT (like abnormal delays, availability or scalability issues, or failover) when the number of emulated users is increased. A load test defines real life like volumes of transactions to test system stability, limits or thresholds. Typically, a number of emulated users interacting with the SUT have to be created and managed while the functional behaviour of their communication with the SUT has to be observed and validated.

M.Ü. Uyar, A.Y. Duale, and M.A. Fecko (Eds.): TestCom 2006, LNCS 3964, pp. 177 – 196, 2006.
© IFIP International Federation for Information Processing 2006

Scalability testing is a special kind of load testing, where the system is put under increasing load. *Robustness testing* is load testing over extended periods to validate an applications stability and reliability. *Stress testing* is the simulation of activities that are more "stressful" than the application is expected to encounter when delivered to real users. Stress tests measure various performance parameters of the application under "stressful" conditions. Examples of stress tests are: *spike testing* (short burst of extreme load), *extreme load testing* (load test with huge number of users), *hammer testing* (continuous sending of requests). *Volume testing* is the kind of performance test we run in order to find which volume of load an application under test can handle.

TTCN-3 (Testing and Test Control Notation) [2] enables systematic, specification-based testing for various kinds of tests including functional, inter-operability, integration, load, robustness, volume and stress testing. It allows an easy and efficient description of complex distributed test behaviours in terms of sequences, alternatives, and loops of stimuli and responses. The test system can use a number of test components to perform test procedures in parallel. The task of describing the dynamic and concurrent configuration is easy to perform since it is developed at a platform independent level. The advantage of this approach is that the distribution configuration is abstract and it does not depend on a particular test environment. The same (potentially distributed) test specification can be executed on different hardware environments and various distribution setups. For example, a test case which creates a number of N test components can be distributed on 5 hosts, but can run also on 10, 2 or just 1 host.

The test workload definition belongs to a performance test plan. It is a description of the test actions against a tested system and should reflect how users typically utilize that system. Overloading an SUT with a huge number of requests tells us how robust the system is, but this kind of test does not reflect normal performance requirements and gives no information about the behaviour of the system in daily scenarios. The workload definition should describe performance tests according to real world scenarios taking into account social, statistical, and probabilistic criteria.

Test distribution is a technique to realize the load as required by the workload definition on several test nodes. Only one test node might not be enough to emulate a big number of users. A distributed test case may consist of two or more parts that interact with each other, but each part is being processed on a different test node.

We concentrate our study on developing and executing distributed load tests with TTCN-3. TTCN-3 offers the required flexibility in specifying load tests. Quite a number of language concepts help to design complex workloads in an intuitive way. However, the real distribution and deployment of the executable tests is out of consideration of TTCN-3. Therefore, an additional layer of specifications is needed to describe the real test configuration on a real target network of test nodes (being potentially only one test node).

This paper discusses in Section 2 related work, and presents in Section 3 foundations like load test specification, test component distribution and factors which influence the distribution. Next, in section 4 the architecture for load tests execution is presented. Section 5 presents the distribution algorithms and discusses their characteristics. Section 6 presents an example. The paper is concluded by a summary.

2 Related Work

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

The first experiments with TTCN applied to performance testing were done with the version 2 of TTCN language. PerfTTCN [5] is an extension of TTCN-2 with notions of time, measurements and performance. In [7] SDL specifications are used to generate tests for distributed test architectures. This paper discusses also concepts related to distributed and concurrent testing. TimedTTCN-3 [9] is a real-time extension for TTCN-3 that supports the test and measurement of real-time requirements. This paper introduces concepts like absolute time, definition of synchronization requirements for test components and provides possibilities to specify online and offline evaluation procedures for real-time requirements. Most of these ideas can be also reused in performance testing with TTCN-3. The work in [4] presents a number of patterns in specification of distributed tests. It uses also TTCN-3 to specify distributed tests and discusses different facets of distributed testing. The test architecture topic is discussed also in [8] where a generic test architecture is presented. As far as the execution of distributed tests is concerned, [6] introduces TCI (Test Communication Interfaces) and discusses the possibility to use TCI to realize distributed test execution environments.

Our paper uses principles of these works and analyses in particular, how test components for load test scenarios can be efficiently specified, distributed and executed.

3 Foundations

Test distribution with TTCN-3 implies, on first hand, the use of the language elements to describe distributed load tests and, on the second hand, the development of an execution environment capable to distribute tests. In this section, we investigate first the TTCN-3 language capabilities to specify load tests along small examples. In additions to this, we look into possible patterns the tester may use for specifying load tests and analyse which are the constraints with respect to the distribution strategies for those patterns.

3.1 Load Test Specification with TTCN-3

TTCN-3 offers various concepts to design load tests such as test components to emulate SUT users/clients, ports to handle connections to the SUT, send/receive or call/reply statements to communicate with the SUT, timers to measure the responsiveness. These concepts are introduced along with small examples of how they are useful in load testing.

`component` is the structural element which is used to define the clients involved in the load test scenarios. One test may define more than one type of components in order to distinguish users of different categories or scenarios.

```
type component UserType {
  port ConnectionType connection;
  timer respTime;
  var integer fail := 0;
}
```

The specification of all test components, ports, connections and test system interface involved in a test case is called *test case configuration*. Every test case has one Main Test Component (MTC) which is the component on which the behaviour of the test case is executed. The MTC is created automatically by the test system at the start of the test case execution. The other test components defined for the test case are called parallel test components (PTC) and are created dynamically during the execution of the test case. The tested entity is called System under Test (SUT) and the interface to communicate with it is the Abstract Test System Interface (`system`).

The behaviour of a test component is defined by a `function`. A function is used in load tests to specify client activities within a test scenario. An SUT client may behave in different ways when interacting with the SUT, thus the test system may have different functions emulating different client behaviours.

```
function clientBehavior(in integer cid)
runs on UserType {
  // user behavior
  // communication with SUT
}
```

TTCN-3 supports message-based and procedure-based communication. The communication operations can be grouped into two parts: stimuli which send information to the SUT (`send, call, reply, raise`) and responses used to describe the expected reaction from the SUT (`receive, getcall, getreply, catch`). To apply a sending operation (stimuli) there shall be specified a port used to send the data, the value to be transmitted, and optionally an address to identify a particular connection if the port is connected to many ports. Additionally, for procedure based communication the response and exceptions are needed; they are specified by using the `getreply` and `catch` operations.

```
connection.send(aRequest(uid));
respTime.start();
alt {
  [] p.receive(correctResponse(uid)) { }
  [] p.receive { }
  [] respTime.timeout { }
}
```

Timers are a further essential feature in the development of load tests with TTCN-3 in order to evaluate the performance of the SUT. The operations with timers are `start, stop, read` (to read the elapsed time), `running` (to check if the timer is running) and `timeout` (to check if timeout event occurred). The start command may be used with parameter (the duration for which the timer will be running) or without parameter (when the default value specified at declaration is used). For load testing purpose, we define timers on test components and use them in the test behaviour to

measure the time between sending a stimuli and the SUT response. If the SUT answer does not come in a predefined period of time, the fail rate statistics should be correspondingly updated.

Another important mechanism provided by TTCN-3 is the inter-component communication which allows connecting components to each other and transmitting messages between them. This mechanism is used in load testing for synchronization of actions (i.e. all components behaving as clients start together after receiving a synchronization token) or for collecting statistical information at a central point.

The handling of verdicts in load tests is different from the traditional verdict handling procedure in functional testing. In functional testing, we use the build-in concept of *verdict* which is always set when an action influences significantly the execution of the test (for example, if the SUT gives the correct answer we set the verdict `pass`; if the response timer expires we set the verdict `inconc` or `fail`). Load tests have also to maintain a verdict which should be presented to the tester at the end of a test case execution. However, the verdict in this case has rather a statistical meaning than only a functional one: Still, the verdict should be a sum of all verdicts reported by client test components. In our approach, the verdict is set by counting the rate of fails during one execution; i.e. if during the test more than a threshold percentage of clients behave correctly we consider the test passed. The percentage of correct behaviours in a tests must be configured by the tester himself and must be adapted to each SUT and test separately.

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

3.2 Load Test Specification Patterns

Test patterns are generic, extensible and adaptable test definitions. Reusable test patterns are (as an analogy of software patterns [13][14][15]) derived from test methods, test solutions and target system technologies. They are available in form of software libraries and/or code generators which offer the tester ready to use code.

3.2.1 Workload Unit Specification

Even though the TTCN-3 language is very flexible and allows for various ways to write a test, we believe that load test designers follow at least one of the specification patterns presented in the following. The major role of a load test is to emulate the parallel behaviour of multiple clients (or users) interacting with the SUT. In literature, the SUT's clients are also called WLUs (workload units) and they are implemented as parallel processes or threads. Nevertheless, a parallel process may emulate the behaviour of more than one user at a time. In TTCN-3, the test component is the building block to be used to emulate one or more WLUs at a time. The parallelism is realized by running a number of test components concurrently on a number of test nodes. The methods to specify user behaviours as test components can be classified into the following patterns:

(a) The most obvious pattern to define a client is to define a component emulating only one client, i.e. the *one client per component pattern*. On this component we start a function which describes the actions the client interchanges with the SUT. Despite the easiness to write load tests using this technique, two main drawbacks exist. Firstly, load control is difficult to realize when the tester wants to keep a constant number of parallel users. The controller needs to control continuously the number of component acting in parallel and whenever a component terminates a new one has to be created. Secondly, the creation, start and termination of components are very expensive operations with respect to CPU on a test node. Because of this, it is preferable to *reuse* the existing test components to emulate more than one client on one test component. But, as we will see in the next section, from the distribution point of view the one-client-per-component specification style turns out to be an advantage for the application of a large number of distribution strategies since the distribution unit is small and the balancing of the load can often be reconfigured.

(b) Another pattern for the specification of load tests is the reuse of components to emulate a new client once the current client terminates. This pattern implies that one component repeats sequentially in a loop the behaviour of a client, but for each client a new set of data (id, request data, client reaction times etc) is used. This pattern, named *sequential repetition of clients per component pattern,* has the advantage that only a fixed number of test components are created and thus no additional time is spent on handling the test components. The disadvantage of this approach is that a test component can be distributed only once at the beginning and no further (re-)balancing is possible.

(c) An extension of the previous pattern is the interleaving of more than one client on a test component. In this way, a test component is able to simulate in parallel a number of clients. This pattern has the name *interleaved client behaviours per component pattern*. Unfortunately, the mixture of parallel behaviours on one component is complicate to specify and most of the time the TTCN-3 code loses its readability and becomes difficult to maintain. The approach has the same disadvantage as the previous pattern that the component can be distributed only once, at the beginning and no further (re)balancing is possible.

3.2.2 Differentiate Component Types

The different client types are usually defined in TTCN-3 as distinct test component types. This is in fact a recommended pattern in test specifications which helps to recognize easier the different types of components at distribution time. Most of the variables used during a test are usually defined directly as part of the test component type so that they can be directly accessed from any function being started on a test component of that type. This approach helps also at distribution time since at the instantiation of a test component most of the memory required by the test component is known right at creation time.

3.2.3 Test Architectures

With respect to test architectures, we identify at least two specification patterns of how to group different kinds of clients interacting with the SUT:

(a) Most of the load tests create independent client components which depend only on the interaction with the SUT. These components do not depend on other components and therefore their distribution on different test nodes is by no means constrained.
(b) Another category of test architectures requires pairs of clients or caller-callee clients to interact with the SUT. Typically, these types of tests need extra communication for coordination between the caller and callee component. Therefore, for these types of tests, the distribution strategy should consider that it is more efficient to install both components on the same test node since the local communication is faster than the communication between two test nodes.

3.2.4 Load Control

In order to control the volume of the created load, the test case has to control the number of users running in parallel. Such a mechanism is called *load control* and it is usually implemented as a separate test component (most of the times it is the MTC) which interacts with all other test components in order to increase or decrease the number of interactions with the SUT. The load is controlled by increasing/decreasing either the number of test components or the number of users emulated by one component. In both cases, the increase of load will bring additionally more need for hardware resources and therefore, increasing the number of components or users emulated after a certain level of the load, the test system will not be able of increasing its load but rather decrease it. Therefore, the tester should take care about this aspect when tuning the test and observe continuously the load of the test system. If it happens that the test system reaches the maximum producible load, then the tester should either try a more efficient distribution strategy for the test components or upgrade/extend the hardware resources.

3.2.5 Synchronization

The synchronization of the parallel test components is realized by passing coordination messages. In general, load tests require synchronization only at the start or stop of the parallel components and/or at increasing or decreasing the level of load. Moreover, the synchronization does not have constraints with respect to the time needed to realize the notification of all test components. TTCN-3 allows the tester to connect all parallel test components, participating as workload units, to a central component (i.e. MTC) which coordinates the synchronized activities.

3.3 Factors Influencing Test Distribution

Resource sharing (CPU, memory, disk or bandwidth) in parallel and distributed computing has been intensively researched over the last decades being the activity of efficient utilization of computing resources by partitioning and balancing the computational load among computing nodes [1]. Load distribution is the strategy to allocate parts of a bigger task to parallel workers (computers or processors) and, thus, to decrease the execution time of a program. Many algorithms have been researched

and applied to particular problems. Depending on the problem, the algorithms work better or worse.

Very often, the parallel processes communicate with each other. Granularity is a parallelism measure which characterizes the inter-process communication. We say that the parallelism has big granularity in case of rare communication or has a fine granularity in case of high frequency of communication between processes.

The class of operations to be performed by parallel processes is a further factor which influences the performance of balancing algorithms. A non-exhaustive overview of classes of operations may classify them into: computational operations (i.e. floating point operations), memory access operations, operations with databases, files operations or communication with other computers.

Synchronization of activities of parallel processes is often needed. A process can be considered a sequence of atomic actions where each action transforms the state of the process. Some of these actions have to be synchronized with actions of other processes. Depending on the used synchronization mechanism, balancing algorithms may perform better or worse. For example, the clock synchronization in difference to message-passing based synchronization avoids the overhead added by the inter-process communication. If the message-passing synchronization is applied, the balancing algorithms should be aware also about the bandwidth consumed for synchronization.

Load testing of hardware components or applications is a resource consuming process which coupes also with the resource sharing discipline. Most of the times, in order to run high performance tests against a hardware component or an application (or just parts of it) many computers have to be involved in the test process so as to create enough traffic to evaluate the behaviour of the SUT under load conditions. In this respect, the tester has to be aware about the possible distribution algorithms and be able to decide which algorithm to apply.

For the distribution of TTCN-3 test components, a number of factors have to be considered when selecting the distribution algorithm. The distribution unit used for test distribution is the parallel test component which can simulate the behaviour of one or more clients. If the component emulates only one user, the component is relatively small and terminates after execution of the test scenario. This design pattern presents the advantage that the balance of resources can be performed at each component instantiation. The creation of components happen at small intervals of times since when a component terminates a new one is created in order to maintain the same number of users. If the component emulates sequentially or interleaved behaviours of more than one user, the component will live for a very long period of time (sometimes until the end of the load test). In this case, the algorithm does not have too much flexibility to balance the load except the creation of the component. In such situations, the recommended strategy should be based on resource consuming predictions.

The existing load on the underlying hardware is a further factor to be taken into consideration when applying a distribution strategy. The solution we foresee for distribution of component targets deployment of load tests on heterogeneous hardware which besides the test application may also run other tasks. Therefore, a continuous observation of the hardware usage is recommended while the distribution strategy considers the resource availability at any component instantiation.

4 TTCN-3 Test Distribution Realization

4.1 Test Component Distribution Language

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

$$h = D(p1, p2, ..., pn)$$

There are two types of parameters which are taken into consideration when distributing test components: *external parameters* like bandwidth, CPU, memory and *internal parameters* like the number of components, type of components, type of behaviours, connections. The external parameters are application independent parameters whose values depend on the execution environment and are constant for all applications running on that environment. The internal parameters are related to the test component based application itself and are different for each test case.

Unfortunately, in TTCN-3 it is not possible to recognize a component by its id. This problem appears when creating test components like in the following example[1]:

```
for (i := 0; i < 100; i := i + 1) {
      var PTCType c := PTCType.create;
      map(c:port1, system:port1);
      c.start(someBehavior1());
}
```

In this example, the component variable c refers to the currently created test component, but is overwritten at each `create` operation, so that the execution environment has no differentiation of the test components. But there are some other characteristics of test components in TTCN-3 which can be used during execution to identify them. These characteristics are of two categories: *behaviour independent* and *behaviour dependent*. The behaviour independent ones concern parameters which can be accessed at the creation phase of the test component: the component type, the instance number and the port types which belong to that component. The behaviour dependent characteristics imply the use of characteristics of the test component we can gather after the component is started or executed (i.e. which Id will receive the test component from the SUT). The distribution mechanisms used in this case are based on analyzing the TTCN-3 code before starting the execution and decide upon execution monitoring where the test components should be deployed. This approach requires running a calibration behaviour in which an instance of a test component is created and its execution is monitored. The observed information is then used during the "real" test in order to decide where to distribute that test component.

[1] Please note that the new version of TTCN-3 being approved summer 2005 offers the assignment of explicit names to test components during their creation, however, this was not available for the presented work.

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

The next XML code is an example of a component assembly file. The `special` tag indicates the host where the MTC component is deployed. The `selector` defines a filter to select all components of type `ptcType`. The selected components can be deployed either on `container1` or on `container2`. One can define deployment constraints for each container (for example, do not allow deployment of more than 100 components on `container2`). The user can also constrain the memory usage, the CPU load, the number of components etc.

```
<component_assembly>
    <description>Example to use TCDL language</description>
    <special container="container1"/>
    <set>
        <component_selectors>
            <componenttype>ptcType</componenttype>
        </component_selectors>
        <homes distribution="round-robin">
            <container id="container1">
                <max_components>10</max_components>
            </container>
            <container id="container2"/>
                <max_components>100</max_components>
            </container>
        </homes>
    </set>
</component_assembly>
```

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

The code below shows a set which deploys the components of types `ptcType2`, `ptcType3` and the instances `1`, `2` and `5` of type `ptcType4` on the `container2` and `container3`, according to a round-robin algorithm.

```
 <set>
        <component_selectors>
            <componenttype>ptcType2</componenttype>
            <componenttype>ptcType3</componenttype>
            <instance type="single">
                <componenttype>ptcType4</componenttype>
                <number>1</number>
                <number>2</number>
                <number>5</number>
            </instance>
        </component_selectors>
        <homes distribution="round-robin">
            <container id="container2"/>
            <container id="container3"/>
        </homes>
</set>
```

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

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

4.2 TTCN-3 Architecture Design for Distributed Execution

For deploying and executing distributed tests, we have designed and implemented the architecture depicted in Figure 1. This architecture follows the ETSI standard architecture [10],[11] for realizing distributed tests . The platform consists of a set of interacting entities which execute the code generated from a TTCN-3 specification, realize the distributed communication between test nodes, realize the communication with the SUT, implement external functions and handle timer operations.

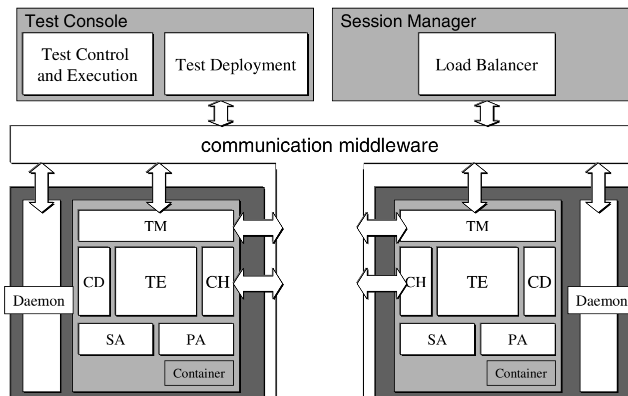

Fig. 1. Distributed test architecture

The Test Console handles the management operations to create test sessions, deploy test components into containers and control the test execution. The tests are deployed, configured and executed in the context of a test session. One of the most important functionality of the session manager is the load balancing one, which coordinates the distribution algorithms (compute the hosts of the components according to assembly rules, performance requirements, distribution algorithms etc). To supply the dynamic algorithms with the necessary information for distribution computation, the Session Manager provides an interface to the daemons in order to gather the resource consuming level.

Test Daemons are standalone processes installed on any hosts which manage the test containers. Containers intercede between Test Console and test components, providing services transparently to both of them, including transaction support and resource pooling. The containers are the hosts of Test Executable; they manage installation, configuration and removal of the parallel test components. Moreover, containers are the target operational environment and comply with the TCI standard for TTCN-3 test execution environment. Within the container, we find the specific test system entities: TM (Test Management), CD (Coder-Decoder), TE (Test Executable), CH (Component Handler), SA (System Adapter) and PA (Platform Adapter). For more information on the API and interactions between these entities, we refer [6]. The container subsystems are functionally bound together by the TCI interfaces and communicate with each other via the CORBA platform.

The distributed handling of the test components is realized within CH. CH distributes TTCN-3 configuration operations like create, start and stop of test components, the connection between test components (connect and map), and inter-component communication like send, call and reply among two TTCN-3 executables participating in the test session. The CH is not implementing the core TTCN-3 functionality – this is done by the TE, for example a test component is created, etc. Next, CH asks the Session Manager for a location for the new component. Based on the decision of the Session Manager, the request for the creation of a component will be either transmitted to the local TE or to a remote participating one if the component has to be created on the remote TE. The remote TE will create the TTCN-3 component and will provide a handle back to the requesting (local) TE. The requesting (local) TE can then operate on the remote created test component via the component handle given by the remote TE.

Hardware load monitoring is used by dynamic algorithms for the balancing decisions. The monitoring tools are running on each host used for the tests and is able to provide to the SessionManager an evaluation of the current hardware consumption. The monitoring tasks are controlled by the SessionManager over a specially designed interface which allow activating/deactivating of different sensors, setting the update refresh rate or counting a performance key parameter out of several parameters.

4.3 Test Execution Evaluation

The intensive use of hardware resources (i.e., 100% CPU) during the test execution leads very often to malfunctions of the test system which ends up running slower than expected. Consequently, the test results can be wrong as an effect of erroneous evaluation of SUT’s responses. We encounter such a situation when, for example, the

test system creates too many parallel processes which share the same CPU. The processes wait in a queue until (according to the used scheduling algorithm) they acquire the CPU. Hence, the bigger the number of processes is, the more time a process has to wait in the queue until it acquires the CPU. Since the execution of critical operations (like timer evaluation, data encoding or decoding, template matching) is automatically also delayed, the test system may consider an operation timed out while, in reality, it did not. The same phenomenon has a considerable impact also on load producing by decreasing the number of interaction per second.

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

One of these parameters is the duration of the execution of critical tasks. We assign temporal dimensions to all operations to be executed sequentially in a test which might influence the evaluation of SUT's performance. For example when receiving a message from SUT and this message is used to validate the reaction of SUT to a previous request, the test system has to decode and match the received message only in a small amount of time, otherwise the additional computation time will be counted as the SUT reaction time. A further interesting parameter is the quantity of the demanded resources. If the test system requires constantly the maximum of the resources the underlying hardware can allocate to them, this is a first sign that the test might not be valid. Another parameter is the deviation average from load shape. If the load does fluctuate very often moving from lower to higher values, it proves that the test system might be overloaded.

We consider that a performance test is valid only if the platform satisfies the performance parameters of the workload. The quality of the load test execution is guaranteed if the test tool fulfils the requirements with respect to execution of the critical operations like decoding, matching, timer processing.

5 Balancing Algorithms Applied to Test Distribution

The literature differentiates load balancing algorithms by several criteria. Load balancing algorithms can be static or dynamic; the difference is made by the distribution decision which is known before actually running the test in case of static algorithms, while the decision depends upon the state of the system when dynamic algorithms are considered. The static algorithms work very well when the test nodes have more or less the same resources (same memory, CPU etc) and the usage during the tests is not influenced by other applications running on that hardware. According to our experience, round-robin algorithm works very well in such situations. The distribution function D, mentioned in section 4.1 is an incremental function over the number of hosts, which selects sequentially the next host for deployment. In the case of test nodes with different capacities, static algorithms do not work well anymore because of hardware limitations. One may try to use round-robin algorithm with empirically chosen constraints for the number of deployed components on each test

node, but this method is difficult to use since the constraints have to be counted any time the number of components is increased. However, the obvious way to handle such hardware configuration is using of dynamic algorithms.

The criterion to distinguish dynamic algorithms is the adaptation to system load. These algorithms monitor and use information about the load of the system before making the distribution decision. The distribution function D is in this case a maximum function over the memory and CPU availability, which selects the next home the one with maximal resource availability. Some of them, the heuristic algorithms even change their policies according to the load of the system; in this case D takes in account a resource consuming threshold. The dynamic algorithms base on thresholds imposed on resource consuming. The threshold can by either the consuming level of a single resource (i.e. memory) or a key performance parameter counted according to a formula which considers several parameters. Dynamic algorithms require, unfortunately, extra activities (i.e., hardware monitoring) on the test nodes, hence the overhead is also bigger than for static algorithms. Also the updates on hardware consuming add some communication overhead when a new component is instantiated. The update on hardware usage may be realized only before a component creation or periodically according to an update rule. The periodical update might be combined with heuristic methods to count the refresh rate, for example the more loaded nodes should have longer delay between updates than nodes which have fewer loads. These algorithms work very well for tests using one client per component since the distribution may take into account the hardware consuming level before any component creation. If the sequential of interleaved behaviour pattern are used, it is recommended to wait a short period of time between component creations until the component reach the average resource consuming level. This approach is based on the assumption that the maximal (or average) resource consuming level for a component remains constant at emulation of sequential clients.

Another category of algorithms are the prediction based algorithms where the decision of deploying test components, formally defined as distribution function D, is based on some preliminary predicted information. For prediction purpose, we have to decide before the start of the test, how many test components to deploy on each node. The preliminary information should also reflect the test component resource consuming. In order to provide this information to the scheduler at the beginning of the test, we should run a small preliminary test to learn something about the behaviour of the test components. From this preliminary test we can measure parameters like: the amount of memory that the test process allocate on each host, the time needed to execute the test behaviour, the maximum amount of memory that a component allocates (the hot-spot). Considering these parameters we may distinguish between two categories of algorithms that can be implemented: memory based prediction algorithms and time based prediction algorithms. Memory management is very important in distributed testing because test components deal with important memory consuming. In this case, running a test on a host which does not provide the necessary amount of memory for test process could lead to a slower execution or a run out of memory exception. For time prediction based algorithms the decision criteria is based on a time factor proportional with the time duration of the component behaviour. To obtain this duration we should measure the time duration of a test component on each node. It is very important to measure the time duration of the same behaviour on each

node. The preliminary test can be executed with one or more components on each node and after every execution an estimated time value (average value) should be profiled from each node. Based on these values for each node, the distribution should be made proportional with the time factor which indicates the number of components deployed on each node.

The control of a load balancing algorithm can be centralized, distributed or semi-distributed. A centralized approach is quite efficient as long the load balancer does not get overwhelmed itself by the request handling task. The distributed approach involves multiple load balancers in decision making. The semi-distributed approaches combine the centralized and distributed approaches; there are several load-balancers which group together multiple server instances and manage them in a centralized way. In our environment we experienced only with the centralized approach since for load testing purpose, the decision making does not add too much computational overhead. We used for our test the sequential or interleaved behaviour specification patterns, which imply that the components are created sequentially. This approach does not necessary require the component to be created very fast, since actually the most important issue is to use efficiently the resources and reach the load level after an undefined period of time.

Depending on the algorithm, the communication overhead is added by the algorithm for distribution and, consequently, the test system requires more resources. In the implementation architecture of the execution platform we presented, the SessionManager is the central entity responsible for the balancing of the test components. We implemented the different distribution strategies within the SessionManager which provides a distribution interface to all test daemons. This interface permits daemons to ask the SessionManager before each test component creation where to deploy that component. Therefore, the distribution operations using the static and prediction based algorithms add only a small communication overhead represented by the request for home location. The dynamic algorithms add a considerable communication overhead since the SessionManager has to be updated by each test node with the level of resource consumption.

6 An Example

In order to experiment with different categories of distribution algorithms we considered a Web server application and designed a load test suite.

The SUT application is a small Web application running on an Apache server. The application presents to its clients two different search forms: for cars and for houses. The information requested by the client is searched in a MySql [12] database. In a typical scenario, as depicted in Figure 2, the user accesses the main page of the SUT. The SUT time, on SUT side, is the time the Web server needs to deliver the main page to the user. The client thinking time on the user side is the time the client needs to read the main page and decide to search a car or a house. When the selection is made, a new request is sent to SUT which delivers back the search form. After another thinking time (to fulfil the form) the user sends the fulfilled search form to SUT. The SUT performs a search in the database and organizes the list of found items in a HTML page. Finally, the result is returned to the client. The search operation can obviously repeat for several times for any client.

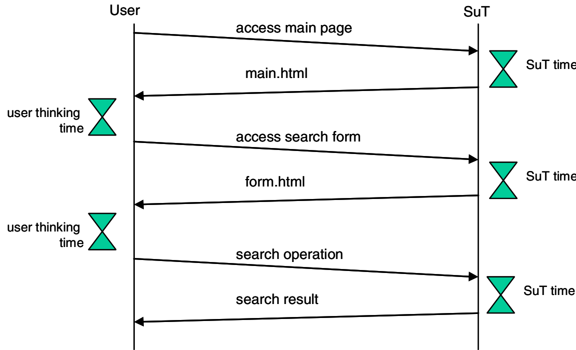

Fig. 2. The sequence of interactions between user and SUT

The design of the load test has the goal to emulate the parallel behaviour of a number of clients which is given as parameter to the test. Any client follows the interaction scenario presented before, but any client has arbitrary thinking times or number of searches in the database. The load (number of requests per second) is controlled by the MTC component which increases or decreases the number of components.

We implemented four distribution algorithms in order to experience with different categories of algorithms.

- *Round-robin (RR).* is a static algorithm which selects the hosts in a sequential order. It proved to be a good algorithm when used on homogeneous environments.
- *Memory threshold combined with round-robin based algorithm (MT).* This is a memory based algorithm that evaluates the percentage of free memory from the Java virtual machine. The decision criterion is based on the available memory for the JVM process on each host and it always deploys a new component on the host with the most available memory. The memory threshold based algorithms used alone could lead to the decision to deploy all components on the same node if the number of components is relatively small and the memory of one host is fairly bigger than on any other host. To avoid situations where all components would be deployed on the same node, the memory threshold algorithm should be combined with round-robin distribution in order to ensure that components will be distributed.
- *Memory factor based (MF).* This algorithm considers the number of test components to be deployed on each host to be proportional with the amount of memory on that host. The rule of deploying components is based on a memory factor that indicates how many components to deploy on each host. For obtaining the memory factor it is necessary to execute a preliminary calibration test for profiling the memory hot-spot.
- *Execution time factor based (TF).* The decision criterion of this algorithm is based on a time factor associated to the behaviour of a client running on a component. To

obtain this duration we should measure the execution time duration of a test component on each node. It is very important to measure the time duration of the same behaviour on each node. The preliminary test can be executed with one or more components on each node and after every execution an estimated time value (average value) should be profiled from each node. Based on these values for each node the distribution should be made proportional with the timeFactor which indicates the number of components deployed on each node in a sequence.

To compare the distribution algorithms we run the load tests and measure the computation time needed by the Test System between receiving a response from the SUT and processing it. This time usually increases with the number of components deployed on the same host. Depending on the algorithm and hardware resources, this time increases differently on the test nodes. The evaluation criterion considers that the best algorithm is the one which makes the computation time stay as small as possible on each node and the computation times grow up uniformly on the test nodes. All the graphs presented next have represented on the vertical axis the time needed for computation and on the horizontal axis the number of components deployed on that host. The test nodes evolution curves are associated with test nodes through dashed lines. The tests are executed on three computers with different hardware resources: `TestNode1` (mem=512Mb, cpu=1.9 Ghz), `TestNode2` (mem=2G, cpu= 2 x 3.5 Ghz), `TestNode3` (mem=1G, cpu=3.5Ghz).

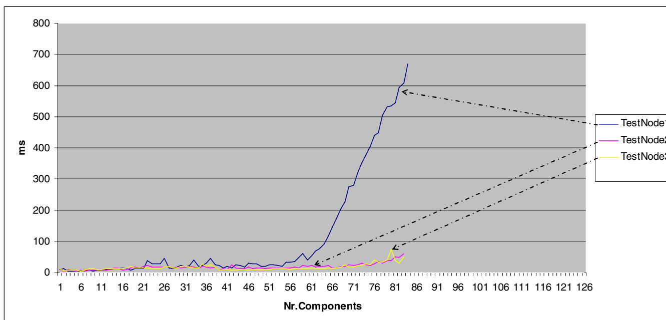

Fig. 3. Execution with Round-Robin algorithm

The Round-Robin algorithm distributes equally the number of components on the three test nodes. We observe that the computation times on the `TestNode2` and `TestNode3` grow very slowly to a negligible value while on `TestNode1` they start growing up after deploying 60 components reaching at the end a computation time of 700ms. This delay may considerable influence the behaviour of the test making possible that some timers will timeout due to a long delay at processing the information received from SUT.

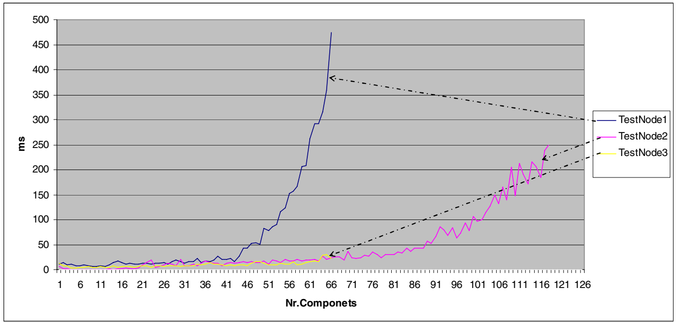

Fig. 4. Execution with Memory Threshold based algorithm

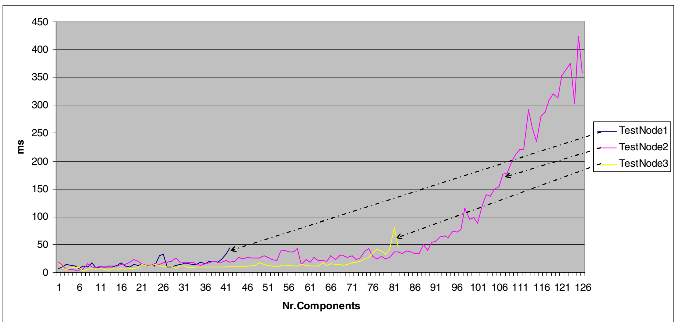

Fig. 5. Execution with Memory Factor based algorithm

Using the memory threshold based algorithm, the test system will deploy a bigger number of components on `TestNode2` so that `TestNode1` will process a smaller number of components. This way the computation time on `TestNode1` will reach only 450ms while on `TestNode2` it will grow now to 250ms. Moreover, one may notice the overhead of the monitoring system on the `TestNode1;` the computation time when using memory threshold algorithm reaches 100 ms after 53 components in comparison to using the round-robin algorithm where 100ms are reached first after 64 components.

A better result is obtained with Memory Factor based algorithm which will deploy a very small number of components on `TestNode1` and `TestNode2`. `TestNode3` will process 126 components but because of its good hardware resources the computation time will reach only 450 ms.

Even better results, are obtained by using the Time Factor based algorithm, which puts more components (almost the same number) on `TestNode2` and `TestNode3` so that `TestNode1` remains with a small number of components. This strategy affects again the performance on `TestNode1` but the computation time reaches only 400 ms which is less than the maximum obtained by the other algorithms.

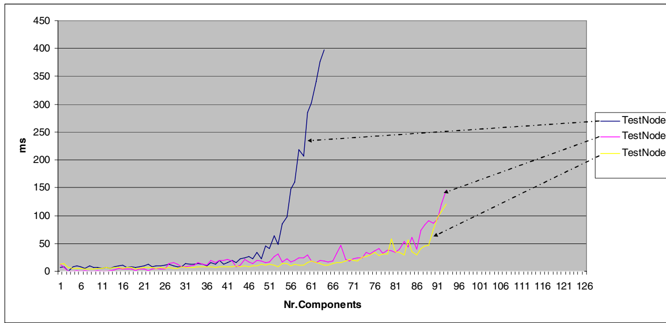

Fig. 6. Execution with execution time factor algorithm

7 Summary

This paper presents a study on applying the TTCN-3 technology for load testing. It introduces the language elements of TTCN-3 which can be used in test specification and discusses several patterns to specify load tests. As far as the execution of TTCN-3 load tests is concerned, the distribution of parallel test components on different test nodes is considered. The distribution is an interesting research topic since many strategies to balance the load can be applied and the balancing algorithms may influence the overall execution of a test. In this respect, we presented various factors which influence the efficiency of test component distribution and discuss different categories of load balancing algorithms. An emerging research subject is to establish theoretically which algorithms are better for special cases of test patterns.

An implementation architecture of the execution environment is also described. In order to experiment with several load balancing algorithms, a load test for a Web server application was performed. The results of experiments show how the distribution strategy influences the overall performance of the test system.

References

[1] B. A. Shirazi and K. M. Kavi and A. R. Hurson, *Scheduling and Load Balancing in Parallel and Distributed Systems*, 1995, ISBN = 0818665874, IEEE Computer Society Press

[2] ETSI ES 201 873-1 V3.1.1, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, Sophia Antipolis, France, July 2005.

[3] ETSI : TTCN-3 Homepage, http://www.TTCN-3.org, June 2005.

[4] H. W.Neukirchen, *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*, Dissertation, Universität Göttingen, November 2004 (electronically published on http://Webdoc.sub.gwdg.de/diss/2004/neukirchen/index.html and archived on http://deposit.ddb.de/cgi-bin/dokserv?idn=974026611)

[5] I. Schieferdecker, B. Stepien, A. Rennoch: *PerfTTCN, a TTCN language extension for performance testing*, in IWTCS'97 Proceedings, Cheju Island, Korea

[6] I. Schieferdecker, T. Vassiliou-Gioles: *Realizing Distributed TTCN-3 Test Systems with TCI*. In TestCom 2003 Proceedings: 95-109

[7] J. Grabowski, B. Koch, M. Schmitt and D. Hogrefe, *SDL and MSC Based Test Generation for Distributed Test Architectures*, In: 'SDL'99 - The next Millenium' (Editors: R. Dssouli, G. v. Bochmann, Y. Lahav), Elsevier, June 1999.

[8] T. Walter, I. Schieferdecker and J. Grabowski, *Test Architectures for Distributed Systems - State of the Art and Beyond*, Invited talk in: Testing of Communicating Systems (Editors: A. Petrenko, N. Yevtuschenko), volume 11, Kluwer Academic Publishers, 1998.

[9] Z.R. Dai, J. Grabowski, and H. Neukirchen. TIMEDTTCN-3 -- A Real-Time Extension for TTCN-3. In I. Schieferdecker, H. Konig, and A. Wolisz, editors, Testing of Communicating Systems, volume 14, Berlin, March 2002. Kluwer.

[10] ETSI ES 201 873-5 V1.1.1: "The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)", February 2003.

[11] Draft ETSI ES 201 873-6 V1.0.0: “The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interfaces (TCI)”, March 2003

[12] MySQL Homepage, http://www.mysql.com

[13] R. Binder: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley, 2000.

[14] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1995