Experimenting with Dynamic Test Component Deployment in TTCN-3

Máté J. Csorba
Department of Telematics
Norwegian University of Science and Technology
N-7491 Trondheim, O.S. Bragstads plass 2B, Norway
Email: csorba@item.ntnu.no

Dániel Eöttevényi and Sándor Palugyai
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
Email: {eottevenyi, palugyai}@tmit.bme.hu

***Abstract*—In our work, we investigate efficient usage of TTCN-3 based test components in the emerging field of distributed performance testing of communicating systems. TTCN-3 is a high level formal specification language used for the specification of Abstract Test Suites (ATSs) suitable for various testing purposes mainly in telecommunication systems testing. A TTCN-3 based test system may consist of several distributed Parallel Test Components (PTCs) and can be executed utilizing diverse hardware platforms. Our aim is to aid development of load tests by establishing a framework, while collecting best practices at the same time, that can be used to execute tests over mixed hardware resources efficiently. We, mainly focus on automated and dynamic deployment of load test components implemented purely in TTCN-3. Also, the deployment method takes test system load into account by monitoring the participating workstations' behavior on-line. A potential test method is shown that can be used for testing server applications using low-cost or even legacy hardware as the platform for distributed test components. A basic scenario involving SMTP (Simple Mail Transfer Protocol) service testing is presented as an example.**

I. Introduction

There are many fields in telecommunications in which development of new protocols and systems is accompanied by extensive testing efforts. Testing and debugging the new mechanisms after implementation is necessary to verify functionalities of the newly created telecommunications system or sub-system. Various test methodologies are adopted in the field of network element design or in the complete domain of communication systems development. Accordingly, a wide variety of testing tools is available in the market that can be used to satisfy testing needs more or less adequately. Lately, telecommunications applications are moving from a hardware intensive approach towards mainly software-based implementations. This fact brings along that testing of communication systems employs software testing methods and more interestingly, common software applications can be utilized to a large extent by the test systems developed. We, in this paper introduce new ideas and guidelines towards a framework for load testing a telecommunications system that uses arbitrary protocols based on a dynamically distributed test environment.

Conventionally, distributed software can be divided to components that implement the actual behavior to satisfy the requirements and a separate part that supports execution of these components. Namely creation, shutdown of components and communication among them, for example. This implies, in most cases, inclusion of an architecture in which a middleware is responsible for component handling operations. A homogeneous component structure on the other hand allows elimination of an additional middleware of different type by implementing the actual distribution and component operation functionalities using the same platform. However, component management functionalities are indispensable in a distributed test system. Useful modules can be added supporting distribution mechanisms if an appropriate interface for the complementary modules exists. Using additional modules the same functionalities can be supported as by the middleware approach only two aspects have to be dealt with. One is the component distribution logic, whether it is static or dynamic it has to be implemented using the same platform, either in a centralized manner or spread across the participating workstations. The other aspect is the on-line measurement and monitoring of system resources, while the test system is executing.

Keeping these fundamental aspects in mind we build a framework based on the Testing and Test Control Notation version 3 (TTCN-3 [1]). Although, the traditional application of the notation was protocol conformance testing that aims to discover deviations in the communication of software or hardware entities compared to protocol standards, with the ongoing development of the notation the focus has widened to allow users to conduct interoperability, robustness and regression testing and more recently load and stress tests also. Similarly to Unified Modeling Language (UML) profiles, extensions have been introduced to TTCN-3 to extend its capabilities such as TimedTTCN-3 [2] for the specification of real time tests. Traditionally, most commercially available protocol testing solutions are either strictly hardware based or they are implemented in a low level development environment. However, there have been case studies and benchmarking approaches published investigating the applicability of TTCN-3 for load testing [3], [4], [5], [6] some of them dealing with efficient test component distribution bringing us closer to our topic. G. Din et al. in [7] define Work Load Units (WLUs) as the elementary descriptors of user load of a service that is tested. According to different patterns WLUs can be assigned to TTCN-3 components depending on the test requirements. In this paper, various parameters of test component distribution strategies are drawn up by the authors classifying the strategies as static or dynamic and centralized or distributed. Also, the mechanisms can be based on diverse runtime parameters, such as memory or CPU consumption, which are to be monitored by the test environment. This again leads to two strategy classes, the ones based on pre-test execution and on resource consumption predictions and the others monitoring resources on-the-fly. It is also identified that there is a need in a TTCN-3 test environment for a load control unit that contains the appropriate logic, belonging to one of the classes defined above, to support test component distribution.

In our approach, in a TTCN-3 based test system two different behaviors are implemented. On one hand, a protocol-specific part is used that communicates with the Implementation Under Test (IUT), on the other hand, test system behavior is implemented in another part of the test environment, each part comprising of several PTCs. Both functionalities are written in TTCN-3, which means that in contrast to ordinary distributed test software emphasis is set on homogeneous component structure, in other words pure TTCN-3 code and implementation.

Combining a distributed component architecture with the flexibility of TTCN-3 allows testing the IUT from several Points of Control and Observation (PCOs) often spread across multiple workstations this way achieving a more realistic test environment. Also, while developing performance critical applications, such as in the case of load testing, with the appropriate component distribution proper load-sharing can be achieved on the participating workstations, sometimes even exploiting different hardware and software platforms hosting the test suite. In addition, further advantages related to TTCN-3 apply to our framework including the platform independent applicability and flexibility of tests. Similarly, high scalability of the test system arises from the careful design of test components and the standard interface among them, which allows easy multiplication of PTCs participating in test. Scalability of this kind also enables test designers to use or reuse legacy hardware elements with relatively limited resources in clusters to test state-of-the-art implementations efficiently, which is one of the main achievements of our paper. Moreover, usage of TTCN-3 PTCs allows inclusion of behavior defined by real finite state machines, specified in an arbitrary protocol specification, into the test components interacting with the IUT, which is compared to script or scenario based tests more accurate and realistic.

The remainder of this paper is organized as follows. In Section II, we present the mechanisms in TTCN-3 that support distributed testing and briefly describe the execution environment that has been used. We detail the behavior of parallel test components and the operation of PCOs. Next, Section III presents generally the guidelines, best-practices and the component structure we set up in our experimental test-bed trying to satisfy the requirements above. In Section IV, an SMTP load tester is presented and details of the execution are shown. Finally, in Section V, concluding remarks are given.

II. Distributed Load Tests in TTCN-3

In this section we give a short introduction to testing with TTCN-3 and with distributed test components in general. Furthermore, the test execution environment we use is also presented.

In most cases, the implementation we test is considered to be a black-box, which means, we do not have information on the internal behavior of the system, only certain responses of it can be caught and interpreted for examination. Accordingly, to test such a system it has to be stimulated externally using protocol messages it is intended to interpret as part of its normal behavior. In a load testing context this means that the test system has to be able to produce the adequate amount of protocol messages and to deliver the messages towards the IUT. Delivery of the proper amount of messages might be limited by the hardware platform underlying the TTCN-3 test environment. Generally, limitations caused by hardware bottlenecks can be solved by setting up even more hardware elements to support a higher amount of running threads. Thus, while testing any kinds of server systems non-sequential behavior of multiple parallel users can be reproduced efficiently by using several processes at the same time.

Either we deal with a workstation having sufficient hardware resources to reproduce multiple users at a time, or in case of tests that utilize the available amount of workstations efficiently, PTCs can be used to assess the performance characteristics of multiple access IUTs. In both cases, the question is what strategy to choose to divide functionalities among the software test components and how to distribute test components efficiently considering the available hardware resources. The correct strategy that we use for distributing test components needs to consider certain important properties of the workstations used and the different capabilities and execution times determined by the inhomogeneity in available hardware. More importantly, the number of virtual users assigned for each workstation must not be left out of consideration either.

A single node might not possess sufficient resources to simulate behavior of the required amount of test users. Also, additional resources are needed to establish communication between test components for registration, coordination, logging or other control functionalities and purposes depending on the implementation of the given test. Internal communication among the components is an important issue while investigating their behavior.

Examination of software components used in testing aims to support their distribution on the available workstations. Distribution mechanisms are either static or dynamic. They are static if they can be run only before the actual test execution, and dynamic if they can work while the tests are executed. Different strategies observe different properties of the hardware/software platform hosting the tests to support decisions regarding distribution of components. Theses properties include but are not limited to average CPU load, memory consumption, frequency of I/O interrupts, network interface parameters. Observation of the background parameters mentioned above can be done statically to supply information for the calculations before the distribution of components, or a framework can be used that is capable of monitoring the underlying platform continuously.

Specifically for TTCN-3, two kinds of approaches are under development so far for static or dynamic distribution. The mechanism can be implemented by providing a middleware platform using another kind of language such as Java [8]. The other kind of approach we propose to use for load tests implies that both the distribution of components and the distribution of load among components is implemented in TTCN-3 together with the test scenarios. By exploiting the full capability of the language itself we can eliminate the bottleneck imposed by an additional middleware. Also, using the proper language elements and because of the platform independent nature of TTCN-3 component handling procedures can successfully be tailored to different hardware environments.

Flexible component handling in TTCN-3 allows setting the test system configuration dynamically. Arbitrary behavior can easily be assigned to a test component, e.g. there can be components communicating directly with the IUT, or components communicating inside the test system only with registration or statistical purposes. Generally, all test components are equivalent, there is only one designated Main Test Component (MTC) that is created in all cases automatically by the executor environment. Thereafter, all the actual PTCs are created run-time by the MTC this way allowing test developers to maintain the component structure.

A test suite defined in TTCN-3 consists of a static and a dynamic part. The static part contains type and template descriptions consisting of component, variable and test port definitions and definitions of incoming and outgoing message structures. Whereas the dynamic part contains behavioral descriptions, functions and test cases. Communication between components, i.e. internally, and communication between test components and the IUT, externally, is completed via test ports implementing the actual PCOs. Components communicate with messages that are stored in queue assigned to each test port until a receive operation is requested by the component. Queues assigned to PCOs can simply be described as FIFOs virtually infinite constrained only by the system memory on the corresponding workstation. However, response times, considerably critical in a load test scenario, depend heavily on queue lengths, this means that the test system needs to achieve as short queue lengths as possible.

Test ports are the protocol specific part of the test environment. Test ports forming the communication channels between the test system and the IUT are the only part of the test system that are written in C++ instead of TTCN-3 and need to be implemented in a relatively platform dependent manner. However, an Application Programming Interface is provided for the test developer to develop test ports rapidly. Importantly, test ports are fairly tight coupled to TTCN-3 test components in contrast to the centralized API approach of TRI [9]. This also bolsters applicability of the TTCN-3 based test system in performance tests [10]. Besides, existing test ports can be re-used in new test configurations as well. Whenever a new incoming protocol message arrives at the test component, an event handler is triggered in the test port and the message is appended to an input queue. Messages enqueued can be extracted by the event handler using the so-called snapshot mechanism [11].

The language supports a specific statement called the *alt* structure that can be used to describe alternative execution paths for behavior such as reception and handling of specified types of protocol messages. This statement, although very useful for easing rapid protocol test development, can be a performance issue also if not applied carefully. Thus, there are different approaches for performance prediction and tuning of test components utilizing *alt* structures [12].

III. Automated Test Component Distribution

Towards a framework for the development of efficient and scalable load test systems we formulate the following guidelines and introduce the building-blocks we propose. The heart of the system is the Load Control module containing the logic handling the PTCs and orchestrating the test campaign. The centralized Load Control logic is located in the MTC, however, it is possible to place it in a separate PTC on a designated workstation if this is necessary for efficient utilization of hardware resources. Internal communication channels, i.e. internal to the test system, are established between the MTC and the PTCs and optionally between the MTC and the IUT itself also (Figure 1). Stimulation of the IUT is achieved via a standard network layer, whereas the protocol of communication with the IUT is dependent of whichever subsystem the test campaign addresses and is incorporated into the test ports of the PTCs.

Beside the Load Control function workstations participating in test might carry two different types of modules, PIPE modules and Load Test Components (LTCs), each of which stays connected with the MTC. LTCs and PIPEs can be started, restarted once running and killed by Load Control and their configuration can be changed run-time also via internal communication. Most importantly, the appropriate logic in MTC allows these operations to be done dynamically after the test campaign was already started.

PTCs might be distributed across an unrestricted number of workstations reasonably depending on the available hardware resources. The allowed operating systems are ranging from any Linux/Sun Solaris UNIX systems to MS Windows environment. LTCs are the protocol dependent building-blocks of the test system containing the actual business logic

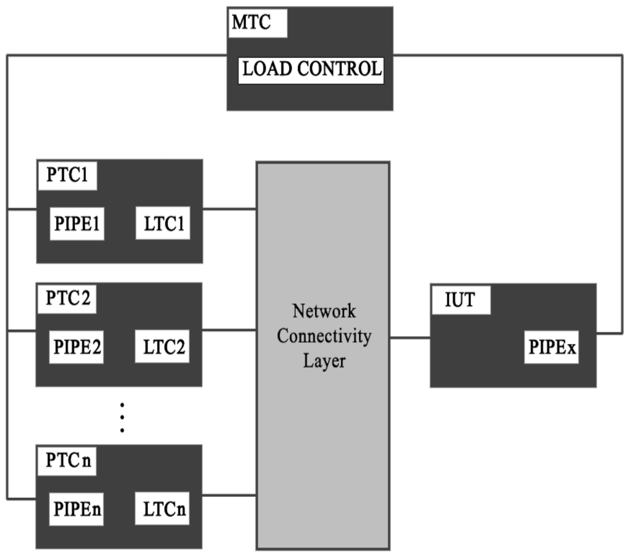


Fig. 1. General structure of test components

to test a specific service of the IUT. LTCs connect via the network layer to the IUT using protocol specific test ports. Also, LTCs might contain finite state machines or basic behavior scripts describing a user who is using the designated service of the IUT.

PIPE components, on the other hand are not communicating with the world outside the test system. PIPE components are used for monitoring the test system internally and for reporting to the Load Control module using so-called Pipe test ports. Pipe test ports are basically connections to the operating system's user plane and providing functionality similar to a user shell that can be exploited running run-time queries or external programs in user space.

The easiest way to gather information on a workstation's resources using a PIPE component is to call external monitoring programs available built in the underlying operating system such as *top, procinfo, ifconfig, ifstat, netstat* with the appropriate parameters. Using a PIPE component information can be extracted at any host participating in test and possibly running a load test component too. Monitored information includes but is not limited to CPU idle time, CPU and memory consumption monitoring for running processes, number of interrupt requests for any connected device, network interface statistics and errors, network connections and opened ports, etc. Optionally, a PIPE component might also be installed on the IUT itself (component PIPE*x* in Figure 1) allowing a very accurate load shape to be produced by monitoring the load directly on the IUT.

In contrast to PIPE components the number of running Load Test Components on each workstation is usually more than one and is regulated by Load Control dynamically. The actual protocol dependent behavior is executed by LTCs this way stressing the IUT. The most obvious setup is when every LTC corresponds to a single test user using service provided by the IUT or one of its subsystems. According to this setup the number of simulated user entities can dynamically be regulated to form the load shape required by the tester. Otherwise, it is also possible to implement LTCs that do not have to be recreated every time their operation ends, only a restart is needed, and of course an LTC can run continuously too with run-time changeable parameters.

Load Control running in MTC is responsible for processing data collected by PIPE components, one for each workstation, and managing the actual LTCs on the available hardware. The number of components running at each time instance is regulated based on the load, whichever aspect of it we are interested in, of the workstations running the LTCs and might also be influenced by the actual load of the IUT. Components can be started and stopped in regular time intervals or instantly depending on the requirements. In the current Load Control logic testers can set the target load at the IUT that has to be achieved. According to this value the MTC creates an initial amount of LTCs and starts running the load test. Thereafter, it checks periodically the test system's and IUT's load and starts or kills LTCs accordingly. More importantly, load balancing is done automatically among the tester workstations, which is especially important in case of legacy hardware with very diverse setup. Each time Load Control decides a new LTC has to be created it checks first, which workstation has the most idle time and deploys the new component accordingly. Shutting down an LTC is done similarly from the most loaded workstation first. This way efficient resources utilization is achieved in the test system.

Beside the number of components, load can be influenced with run-time parameters of LTCs as well. The configurable and dynamically reconfigurable parameters include the initial amount of test components, the number of running components at each host, the time interval between two component creation operations to achieve a uniform load shape, the poll time interval the test system uses to monitor the underlying hardware platform regularly. Last but not least, there is a dynamically configurable parameter called *dt* that represents the elementary time interval between each message transmission an LTC executes. Thus, using this parameter fine-grained performance tuning of load generation can be achieved at each LTC.

IV. Example Test Scenario

Following the concept in Section III we designed a load test system for SMTP service [13]. We have chosen to use the server application *sendmail* to act as IUT [14]. The configuration of the SMTP server was kept as simple as possible with standard settings.

Although, there are simple SMTP stress test applications existing, such as MultiMail [15], their application is constrained, e.g. only ten concurrent threads are supported. Moreover, no real distributed applications are available, i.e. test execution is limited to a single workstation. However, our aim was not the development of an SMTP test system, i.e. the test

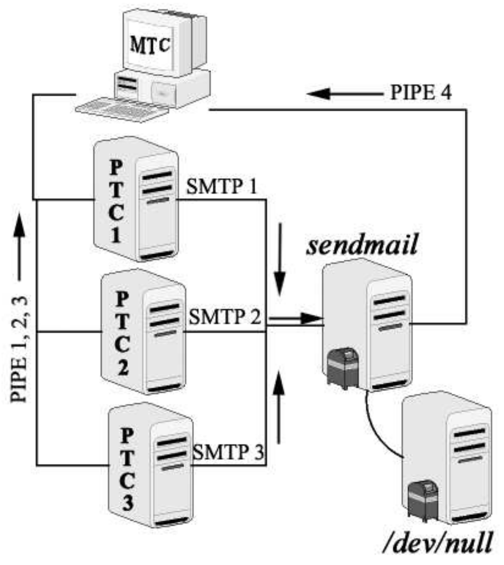


Fig. 2. Example SMTP test scenario

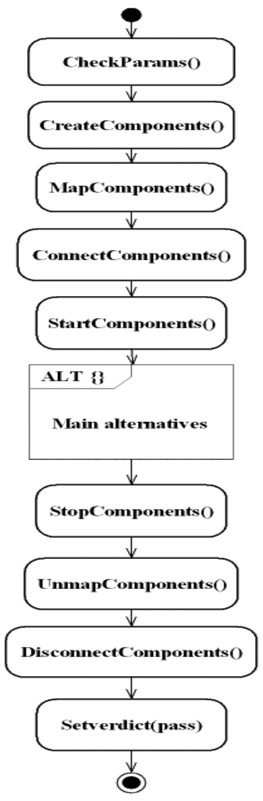


Fig. 3. The behavior of MTC

scenario was only used for demonstrating the idea behind our load test experiment. Accordingly, the simple test scenario in Figure 2 aims to demonstrate effectiveness of a TTCN-3 load test system distributed dynamically using a centralized Load Control logic this way stressing an IP service with significant user load using cheap, legacy hardware and efficient resource utilization.

Two servers were used running SMTP service, one of which was the actual IUT and the other was used only as an endpoint of messages erasing every e-mail immediately after reception. Load Test Components running SMTP protocol simulated the behavior of artificial e-mail clients residing on three different workstations and Load Control within the MTC was running separately on a fourth workstation.

The behavior of the Load Control logic inside the MTC can be seen in Figure 3. After checking the run-time parameters an initial number of components are created and their communication ports are mapped (external test ports) and connected (internal test ports) to the corresponding endpoints. After the components are started the main behavior part of the MTC follows, until the predefined test run period is over (*tExecutionTimer* timed out). Eventually, parallel test components are stopped, their connections are either unmapped or disconnected and test execution ends with a *pass* verdict if everything went fine.

The main behavior of MTC contains seven branches (Figure 4), timer events and reception of messages mixed. Whenever a response is received following the polling of a PIPE component, the response message is evaluated and actions are taken if needed. PIPE response messages carry measurement information on the performance of the test components and the IUT. LTCs can be shut down, started or simply their output can be biased as needed. PIPE components are polled for performance information regularly as set in timer *tPollTimer*. Any unwanted messages are erased by *Trash()*. If a test component quits unexpectedly the test execution's verdict is set to *fail* and the shutdown procedure starts. Similarly, when the tester defined execution time is up, the MTC starts to shut down, however in this case with a *pass*, which cannot override the global verdict if it has been set to *fail* previously. After a LTC has stopped, it sends the amount of e-mails it delivered that is counted globally in *GlobalMailCounter*. The MTC can quit the main *alt* loop gracefully after a quit period indicated by *tQuitTimer*.

The behavior of a PIPE component is shown in Figure 5. The component waits for a poll message from the MTC, which can contain a query command destined for execution by the underlying operating system, alternatively a hardcoded query can also be executed. The query will be answered by the operating system of the workstation via the local Pipe test port and the result will be sent to the MTC using the internal test port. Additionally, for deadlock avoidance a timer is run between each poll and response pair inside the component.

At the same time, in SMTP components, which are the actual LTCs, SMTP communication is performed using a Telnet test port [16], since SMTP protocol uses a simple character string based communication (Figure 6). After logging in to the server, the component starts its main *alt* loop. The SMTP test component evaluates every response received from *sendmail*, checks the response for errors and sends e-mail traffic with variable parameters. The amount of e-mails sent at

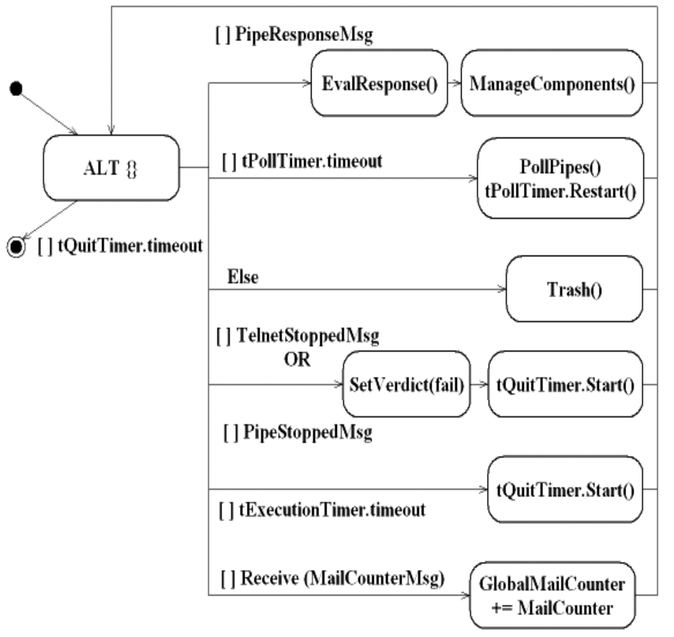


Fig. 4. The main *alt*{} in MTC

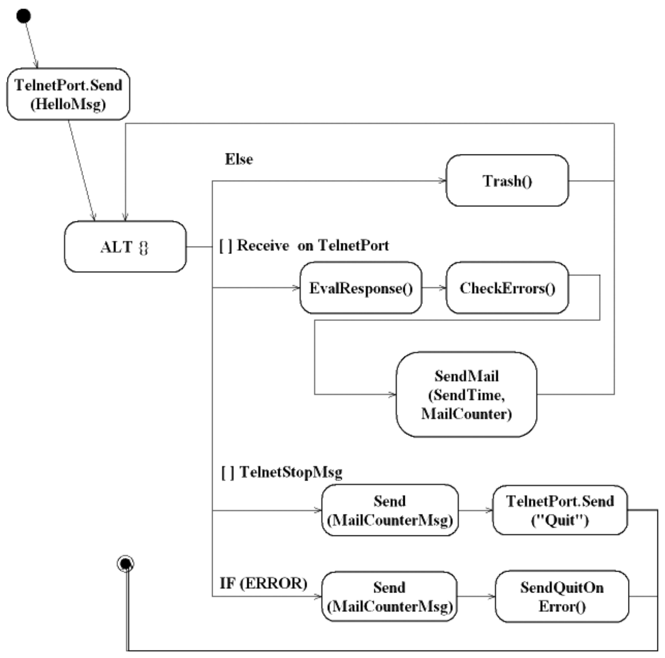
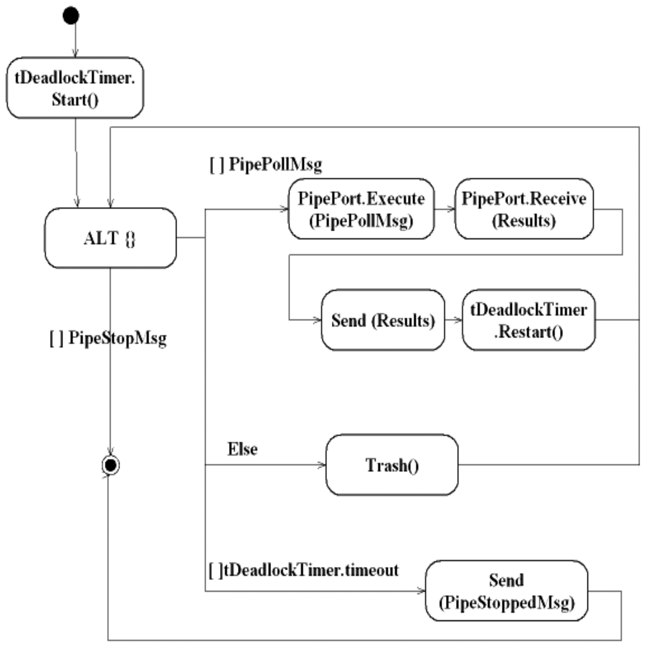


Fig. 6. Simplified state machine of an SMTP test component



tDeadlockTimer. Start()
[ ] PipePollMsg
ALT {}
PipePort.Execute (PipePollMsg)
PipePort.Receive (Results)
Send (Results)
tDeadlockTimer .Restart()
[ ] PipeStopMsg
Else
Trash()
[ ]tDeadlockTimer.timeout
Send (PipeStoppedMsg)

Fig. 5. Simplified state machine of a PIPE component

each call, the size of the payload and the inter-message time can be specified within each individual component this way regulating the test traffic. If the component detects an error in the communication, it gracefully shuts itself down. Whether quitting because of an error or quitting intentionally because the MTC ordered to do so, the component first reports the amount of e-mails delivered and closes the connection towards the SMTP server afterwards.

The workstations' and IUT's hardware was selected to reflect the differences of the environment, performance and resource-wise, the test system is intended to be used in. First of all, IUT's and the supplementary server's hardware platform was an Intel Pentium 4 server with a CPU running at 3.2 GHz equipped with Intel's Hyper-Threading technology [17], with 2 GBytes of RAM and with a Serial-ATA hard drive. The test system on the other hand consisted of legacy PC workstations. To reflect the diversity of hardware we intended to test with, four different configurations were used with Intel Pentium II and Pentium III CPUs ranging from 450 to 600 MHz with memory sizes between 128 to 512 MBytes. The operating system used was SUSE Linux 10 in all cases. Network connectivity was established using 100 Mbps switched LAN, each workstation and server utilizing two Ethernet cards for separating the test system's internal traffic and the SMTP test traffic.

Adhering to the test configuration in Figure 2, using three workstations as load testers and the additional one as MTC, we tested the Intel Pentium 4 server first with Hyper-Threading turned off. In Figure 7, the targeted load level, which is a test configuration parameter, can be seen on the x-axis and the amount of e-mails successfully sent during the test period is depicted at the y-axis. The parameter Target Load represents the load level Load Control in MTC aims to achieve at the IUT by regulating the number of running test components accordingly. Another parameter that influences output of the test system is *dt*, which is the time interval between each message transmission of a LTC. *dt* was chosen 1, 3 and 5 seconds and there was a test letting LTCs put out protocol

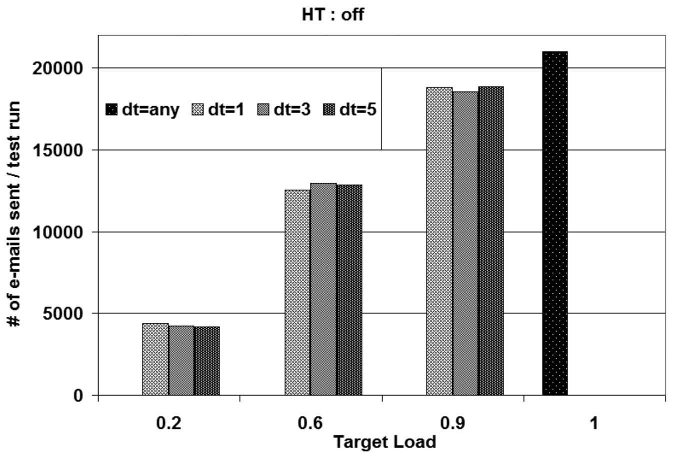


Fig. 7. Test traffic, Hyper-Threading off

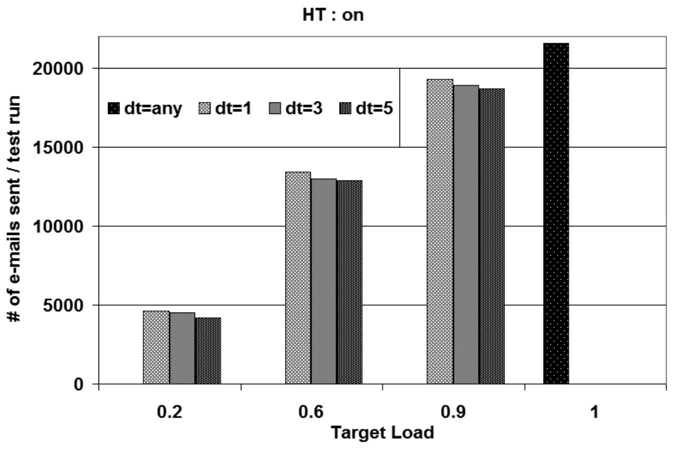


Fig. 8. Test traffic, Hyper-Threading on

messages back-to-back without a delay (*dt=any*).

In case of Hyper-Threading technology is turned off in the server Load Control regulates the number of components in a way that for a certain Target Load the amount of test traffic is approximately equal for every *dt*.

After turning Hyper-Threading on, slightly higher throughput can be achieved at IUT, because system processes can run more effectively nearly as if the server was a real dual-CPU system. Thus lower *dt* represents slightly higher load also (Figure 8).

Test execution time was chosen approximately 25 minutes to allow the IUT a suitable rise time in message transmission rates. According to Figure 7 and Figure 8 in a 25-minute test run approximately a 14-e-mail-per-second throughput could be achieved on the available legacy hardware with the test system. By comparison, an above average mailing list populated by 2,000 subscribers delivers around 400,000 messages per day, i.e. 4.6 e-mails per second [18], [19].

A lower *dt* value means faster transmission of protocol messages at each LTC that involves more resource consumption at the test nodes. Accordingly, Load Control deploys a lower number of components to each of the workstations participating in test. However, this fact also means that in case of a test method that simulates one client at each test component a higher number of clients can be reproduced at the cost of a lower *dt* (Figure 9).

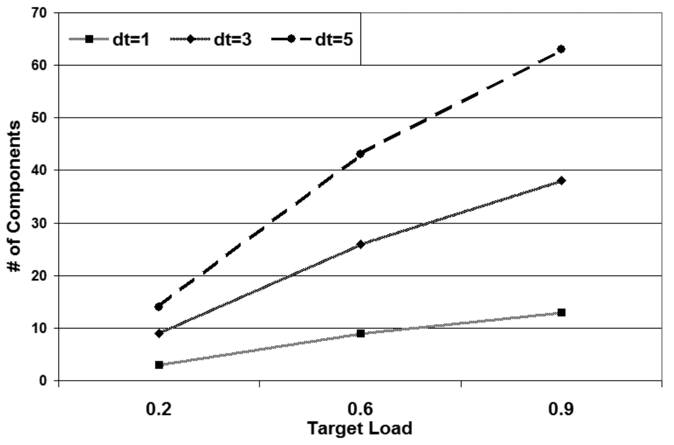


Fig. 9. Number of test components required for different load levels

V. Conclusions

In this paper, we presented a novel approach for designing load tests using dynamically distributed TTCN-3 test components supported by an experimental test setup involving SMTP service testing. Consequently, usage of TTCN-3 in load tests does not imply a bottleneck if tests are designed carefully, but the language elements allow designing an effective distributed load test framework.

The centralized test component distribution logic is easily interchangeable allowing the development of more effective algorithms supported by in-depth measurements or analysis such as in [12] that can be used to predict delays, losses, response times or throughput in PTCs.

The scope of the dynamically configured parameters includes the number of PTCs running, timing of component creation, i.e. the delay between the creation of two components and the message output rate of each test component. Dynamic parameters can be set and reset based on the on-line measurement of memory consumption and of processing speeds at each workstation participating the test campaign. The number of PTCs is regulated by a central Load Control module that aims to produce the load level set by the tester as accurately as possible, this way allowing testing of the IUT's performance with a uniform load shape.

References

[1] 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", 2005.

[2] Z. Ru Dai, J. Grabowski, H. Neukirchen. Timed TTCN-3 - A Real-time Extension for TTCN-3. I. Schieferdecker and H. König and A. Wolisz (Eds.) Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, pp. 407-424, 2002.

[3] S. Dibuz, T. Szabo, Zs. Torpis. BCMP Performance Test with TTCN-3 Mobile Node Emulator. Proceedings of the IFIP 16th International Conference on Testing Communicating Systems - TestCom 2004, pp. 50-59, 2004.

[4] R. Gecse, P. Kremer, J. Z. Szabo. HTTP Performance Evaluation with TTCN. Proceedings of the IFIP 13th International Conference on Testing Communicating Systems - TestCom 2000, pp. 177-192, 2000.

[5] Z. Wang, J. Wu, X. Yin, X. Shi, B. Tian. Using TimedTTCN-3 in Interoperability Testing for Real-Time Communication Systems. Proceedings of the IFIP 18th International Conference on Testing Communicating Systems - TestCom 2006, LNCS 3964, pp. 324-340, 2006.

[6] M. J. Csorba, S. Palugyai, J. Miskolczi, S. Dibuz. Performance Measurement of Routers using Conformance Testing Methods. Proceedings of the 2nd International Workshop on Inter-Domain Performance and Simulation - IPS2004, pp. 152-158, 2004.

[7] G. Din, S. Tolea, I. Schieferdecker. Distributed Load Tests with TTCN-3. Proceedings of the IFIP 18th International Conference on Testing Communicating Systems - TestCom 2006, LNCS 3964, pp. 177-196, 2006.

[8] I. Schieferdecker, T. Vassiliou-Gioles. Realizing Distributed TTCN-3 Test Systems with TCI. Proceedings of the IFIP 15th International Conference on Testing Communicating Systems - TestCom 2003, LNCS 2644, pp. 110-127, 2003.

[9] ETSI ES 201 873-5 (V3.1.1): ”Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)”, 2005.

[10] J. Z. Szabo. Experiences of TTCN-3 Test Executor Development. Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, pp. 191-200, 2002.

[11] ETSI ES 201 873-4 (V3.1.1): ”Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics”, 2005.

[12] M. J. Csorba, S. Palugyai, S. Dibuz, Gy. Csopaki. Performance Analysis of Concurrent PCOs in TTCN-3. Proceedings of the IFIP 18th International Conference on Testing Communicating Systems - TestCom 2006, LNCS 3964, pp. 149-160, 2006.

[13] J. Klensin, Ed.. Simple Mail Transfer Protocol, RFC 2821. 2001.

[14] R. Blum. sendmail for Linux. Sams. 2000.

[15] http://www.codeproject.com/tools/multimail.asp

[16] J. Postel, J.K. Reynolds. Telnet Protocol Specification. RFC 0854. 1983.

[17] E. Severinovskiy. Intel Hyper-Threading Technology Review http://www.digit-life.com/articles/pentium4xeonhyperthreading/

[18] R. Kolstad. Tuning Sendmail for Large Mailing Lists. Proceedings of the 11th Systems Administration Conference - LISA’97, pp. 195-204, 1997.

[19] C. Assman. Sendmail X: Performance Test and Results http://www.sendmail.org/~ca/email/sm-X/design-2005-05-05/main/node6.html