PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 47, NO. 1–2, PP. 101–113 (2003)

# PERFORMANCE TESTING ARCHITECTURE FOR COMMUNICATION PROTOCOLS

János Zoltán SZABÓ[1]

High Speed Networks Laboratory
Department of Telecommunications and Telematics
Budapest University of Technology and Economics
H–1117, Magyar tudósok körútja 2, Budapest, Hungary
e-mail: szaboj@ttt-atm.tmit.bme.hu

Received: Feb. 2, 2003

## Abstract

This paper presents a new model for distributed and parallel performance testing. The model has been designed to support testing based on language Testing and Test Control Notation version 3 (TTCN-3). Our test environment is capable of generating a realistic load towards the tested implementation with a large number of distributed parallel tester processes. The architecture is also able to operate on test systems with heterogeneous hardware and operating systems.

We show the components of our model in details and demonstrate their operation and internal communication on examples. The practical issues of the test system implementation are also discussed. Some results from real life performance testing applications conclude the paper.

*Keywords:* test architecture, performance testing, TTCN-3, distributed testing.

## 1. Motivations

In today's telecommunication protocols and applications it is important to check not only the functional correctness of implementations, but their performance characteristics as well. The purpose of performance testing is to verify whether the tested system can work under realistic load conditions and is able to cope with overloaded situations.

The commonly used protocols exchange complex data units based on sophisticated behaviour rules. Thus it is practically infeasible to set up a test environment only for performance testing from scratch, which is able to communicate with a real protocol entity. Performance evaluation takes place in the last phases of the verification process because it assumes the functional correctness of the implementation. The most suitable way to do performance testing is to re-use some parts from the existing functional or conformance test environment. For example, data type definitions and message encoders/decoders are available from earlier phases.

[1]Some parts of this work were performed in the Conformance Center of Ericsson Hungary's Research and Development division

This implies that, if possible, performance testing should use the same test notation as conformance testing. The most widely used standardized test language in the telecommunication area is the *Tree and Tabular Combined Notation* (TTCN). This paper proposes a new, TTCN-based performance testing architecture and deals with its theoretical and practical issues.

The paper is organized as follows. The next section presents the existing solutions and approaches for performance testing. Section 3 describes our proposed performance test architecture in details. Section 4 deals with implementation issues. Finally some case studies show the applicability of our test environment in practice.

## 2. Preliminaries

The implementations of communication protocols are usually distributed systems. Performance testing includes the checking of functional correctness because the high throughput and short response times are useless if the functional behaviour is incorrect. That is why performance testing requires at least as much computational power for the tester as the tested implementation. Therefore the testing of distributed implementations is feasible only with distributed test environments. This raises significant problems such as the synchronization and coordination between remote testing entities or the interoperability between different hardware/software platforms.

A possible technique for measuring the performance characteristics of end-to-end services and applications is to model each protocol entity with a service user based on a probabilistic timed state machine ([1]). The test environment runs a large number of these simple automata in parallel; each of them is communicating with the tested system independently. When the load generation is finished the test environment collects the measured results from the parallel processes and sets the final verdict.

The original version of TTCN language, TTCN-2 ([2]) was designed for conformance testing of OSI reference model based protocols. TTCN-2 had both theoretical and practical shortcomings when it was applied for performance testing. The most serious theoretical trouble was the static test configuration of TTCN-2.

In the TTCN terminology the processes that are executing test behaviours are called test components. A test configuration consists of two kinds of test components: the only one *Main Test Component* (MTC) and zero, one or more *Parallel Test Components* (PTC). The test configuration model of TTCN-2 is static, what means that the number of PTCs is fixed and each of them shall be separately declared and named uniquely. The connections between two test components and the interfaces toward the *System Under Test* (SUT) are also static.

The practical difficulty with TTCN-2 is based on its tabular format: the test specification consists of several complex and strictly defined tables. Therefore the TTCN-2 test executors are enormous and very complex systems with low execution speed. This makes performance measurements with TTCN-2 unfeasible in most cases.

There were attempts to overcome the difficulties with TTCN-2. One of them was *PerfTTCN* ([3]), a language extension, which defined a new performance test configuration. PerfTTCN groups the test components into two categories: the *Foreground* and *Background Test Components* (FTCs and BTCs). There are only a few pieces of FTCs, but the number of BTCs is scalable. FTCs make just a few sample sessions with SUT and measure its performance characteristics. At the same time a large number of BTCs generate a bulk load toward SUT without measurements.

Our previous trials showed in [4] that PerfTTCN had some disadvantages. The major one was that only the FTC behaviours were described in TTCN-2. The BTCs had to be implemented as external programs in common programming languages like C. This means a lot of extra work and extra cost, and not all TTCN-2 test systems have open interfaces to extend the language. Another drawback is that only an insignificant portion of traffic is monitored thoroughly, failures in BTC communications usually remain hidden.

To abandon such limitations a new revision was made on the TTCN language. TTCN-3 ([5]) has been recently standardized by ETSI. Compared to TTCN-2 the syntax has been simplified, but the application areas have been extended. In addition to the traditional conformance testing, the language can be efficiently used to specify performance test scripts. Moreover, the language allows the derivation of test cases for measuring performance characteristics of protocol implementations easily from existing TTCN-3 conformance test suites by simply re-using data definitions and message templates.

The built-in TTCN-3 language constructs allow to create any number of PTCs during the test run. Not only the number of PTCs may change during test run, but also the test components can be dynamically re-configured. This means, the test components can terminate and re-establish their communication connections with each other or towards SUT.

However, the TTCN-3 standard does not deal with the implementation of performance test systems. The test executor abstract virtual machine is defined in the operational semantics as a centralized singleton process. This paper presents a possible, working solution for the test system realization. Of course, our basic problem is not a recent one; there have already been some existing standardized solutions for distributed testing and test synchronization.

In the next paragraphs we would like to summarize our arguments for defining a new test architecture and control protocols. The test coordination standard that has the most similar purpose to ours is called Test Synchronization Protocol 1 plus (TSP1+, [6]).

The main goal of TSP1+ is to provide a common interface for interoperability between TTCN test equipment of different vendors. The TSP1+ communication primitives cover the constructs of the TTCN-2 only. TSP1+ is unable to handle, for example, the creation of TTCN-3 PTCs and the dynamically changing test configuration because TTCN-3 has no static limits and PTCs have no unique names. On the other hand, TSP1+ was designed for managing conformance test execution for complex systems and not for performance testing at all. Moreover, the TSP1+ messages contain many optional fields to better fit the needs of different tool vendors.

If we had chosen TSP1+ for the control protocol between our test system entities, we would have had to extend it with new TTCN-3 features and we have had to handle a lot of unnecessary options. Thus the final outcome would have been a relatively slow implementation of a standard protocol with proprietary extensions. The execution speed is a key issue in the case of performance testing, so we decided to design a completely new architecture from scratch, which fits our needs the best. We tried to minimize the number of data fields in all communication primitives and we specified the behaviour of all entities to be unambiguous.

## 3. Architecture for Parallel Test Execution

In our proposed architecture the test system consists of several parallel processes and control connections between them. *Fig. 1* shows the set-up of these processes in the case of an example test configuration. The processes of the Test System as well as the System Under Test may be distributed among many computers. Because TTCN-3 uses the black box testing approach, the internal structure of SUT is not important from the tester's point of view. Therefore, in the following we will focus on the different types of components that form the Test System.

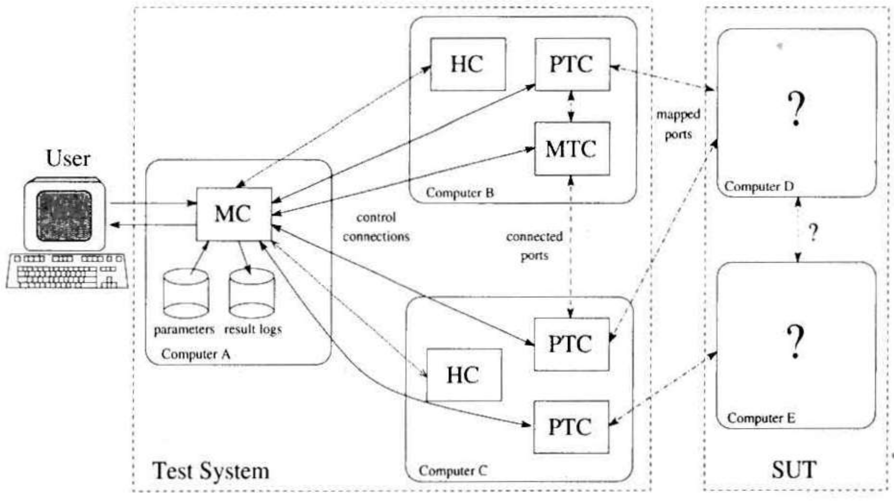

*Fig. 1.* A sample configuration with the parallel test architecture

### *3.1. Elements of the Test System*

The components of Test System can be grouped into three functional units: the *Main Controller* (MC), the *Host Controllers* (HC) and the *Test Components* (TC).

There is one Main Controller in the Test System. Its main responsibility is to perform those tasks that require central coordination, such as assigning a unique identifier for a newly created Test Component. Therefore the MC maintains bi-directional control connections with all other components. In addition, MC provides the user interface for the entire Test System. The user can start or interrupt the test execution and view the test results on a graphical or command line interface. The program code of MC is static, that is, it does not depend on the test suite to be executed.

There may be one or more Host Controllers in the Test System, but there must be exactly one HC on every computer that participates in test execution. The HCs have only one task, to create new Test Components locally on that computer. Whenever a request for component creation arrives from MC, the HC duplicates itself and the child process will become the new TC. Thus the program code of HCs shall include the executable format of the TTCN-3 test suite. To assure the consistent behaviour of the Test System, all HCs (and therefore all TCs) must run the same program code.

### *3.2. Creating Test Components*

Test Components realize the behaviour of each TTCN-3 test component by running the executable form of the test specification. The TCs can communicate with each other or with SUT using abstract messages as specified by TTCN-3 communication primitives (e.g. `send` or `receive`). In the Test System there is one dedicated TC, which corresponds to the TTCN-3 MTC and others are the equivalents of TTCN-3 PTCs. Initially, only the MTC is created so it is the logical ancestor of all TCs. Not only the MTC but any PTC may create new components.

The test execution works as follows. First, the user starts the MC and the HCs on each computer. After that, the MC instructs a HC to create the MTC and starts a TTCN-3 test case on it. The creation and termination of PTCs are fully controlled from the TTCN-3 test suite.

### *3.3. Component Creation*

When the MTC reaches a TTCN-3 create statement below:

```
var MyComponentType MyPTC := MyComponentType.create;
```

it blocks itself and sends a request message to the MC on its own control connection. MC will choose a host and forward the request to the corresponding HC. When the new PTC is ready, it notifies the MC on its new control connection, and finally the MC sends an acknowledgement back to MTC and both components can run in parallel. The Message Sequence Chart (MSC) of the create operation can be seen in *Fig. 2*.

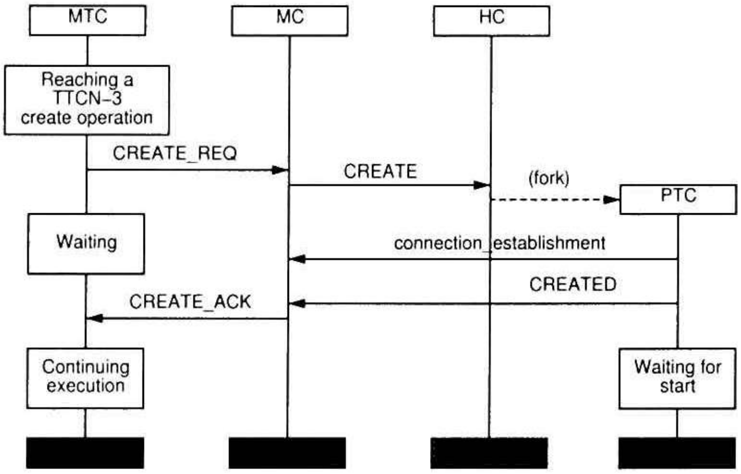

*Fig. 2.* MSC of create operation

### *3.4. Component References*

In a TTCN-3 test system every test component must have a unique identifier, which is called *component reference* in the language. The TTCN-3 standard specifies the component references to be implementation dependent, but those values must behave transparently like in the case of any built-in data type. The test components can store them in variables, pass them as argument to functions or even communicate them to other test components within internal messages. The component references are returned by the create operations. In our previous example the component reference of the newly created PTC is stored in variable named `MyPTC`. The other built-in TTCN-3 component and configuration operations, which operate on already existing components, take one or more component references as arguments.

In our test architecture only the Main Controller sees a consistent picture about the state of test components. Therefore only MC is able to allocate the unique component references. To make the implementation as simple as possible the component references are pure integer numbers, that is, they do not imply the type or physical location of the test component.

The component references of PTCs may be arbitrary; our MC uses a monotonically increasing continuous range for that. The Main Test Component, however, has a special role: it cannot be created or stopped, but it can have port connections. Thus MTC must also have a well-known component reference, so MC uses a fixed number (1, for example) for this.

To avoid confusion the MC must not assign an already used component reference to a newly created PTC even if its previous owner has already terminated. This is because any TC may have stored the reference of the old PTC and may use it in future operations. Although all TCs must terminate at the end of test case execution, there exists a construct in TTCN-3 to define such variables that preserve their values between successive test cases[2]. Therefore the component references must be globally unique during the entire test execution and it is not a good idea to re-start the component reference allocation algorithm in the MC at the beginning of each test case.

## *3.5. Component Location Strategy*

Choosing the appropriate host for a newly created test component is one of the key functions in MC. Our algorithm comprises two steps. First, the set of candidate hosts is determined based on the type of the requested component. In general, the network location and the different hardware configuration of the participating computers do not allow running any kinds of components anywhere. The mapping between component types and hosts is specified with special rules in the test configuration, outside of TTCN-3 test specifications.

The second step uses a load-balancing concept. The host with the lowest load is chosen from the set of candidates in such a way that MC forwards the create request to the computer that runs the least number of TCs. Therefore the overall performance of the Test System is scalable with the number of computers participating in test execution.

## *3.6. Control Protocols*

The control protocols between the test system entities assume a reliable transport layer for all connections. The connections between different entities have different semantics with different messages. The different types of arrows in Figure 1 denote different kinds of connections. Altogether we use around 50 different types of messages with various attributes. During the design of our control protocols we eliminated all redundancies and ambiguities (e.g. options) from the messages to achieve simpler and faster implementation. Our test architecture covers all parallelism related TTCN-3 language constructs, such as PTC creation and termination

[2]An example is when a local variable of the TTCN-3 module control part is passed as `inout` parameter to several test cases.

(as it was shown before) as well as the handling of internal communication channels between PTCs.

The following section shows an example that illustrates the working mechanisms of the control protocols within the test system. We present the implementation of TTCN-3 *connect* and *disconnect* port operations only, the other TTCN-3 operations work in a similar way. In addition, we describe only the normal way of working for these operations. The recovery mechanisms from various error situations are out of scope of this paper.

## *3.7. Handling of Port Connections*

The abstract interfaces of TTCN-3 a test component towards the outside world are called communication ports. TCs can send or receive abstract messages on these ports to or from the SUT or other test components. In the first case the port of a TC shall be mapped to SUT using TTCN-3 operation *map* and *unmap*[3]. Making the physical connection to SUT is the task of the protocol adaptation and is outside the responsibility of the test architecture. In the second case, however, the internal connections within the test system shall be handled entirely by the test architecture.

In order to communicate between two TCs the TTCN-3 test behaviour has to perform a connect operation on a port of each TC. A port connection allows two-way message transfers between two ports. A connection establishment or termination may be requested either from one of the end-points or from a third TC by calling the built in operations connect or disconnect. The TTCN-3 language allows a port to have multiple connections to different TCs at the same time so that each connection shall be handled separately. In our test architecture the port connections are realized by direct transport layer connections between the processes realizing TCs. This ensures the fastest transfer of internal messages to their destinations even in the case of load testing when a large number of internal messages are communicated.

When a TC is created it has only one control connection to MC. Therefore the central coordination of MC is always required for establishing or destroying of a port connection. In our examples we would assume that TC A requests the building and destroying of a port connection between two other TCs (let us call them B and C). *Fig. 3* shows the test system before the port connection is set up. MC also keeps an eye on all existing port connections of the test system. This can avoid inconsistent behaviour if the connection with same endpoints is requested by two different TCs at the same time.

The connection handling protocols are precisely described using Finite State Machines (FSMs). The two endpoints A and B shall behave according to the same FSM but, of course, both have their own instance. In MC there is another type

[3]Execution of map and unmap operations also require the exchange of special control messages, especially when the mapping is initiated by a remote component. These operations are much simpler compared to connect or disconnect, thus we do not present them due to the lack of space and relevance.

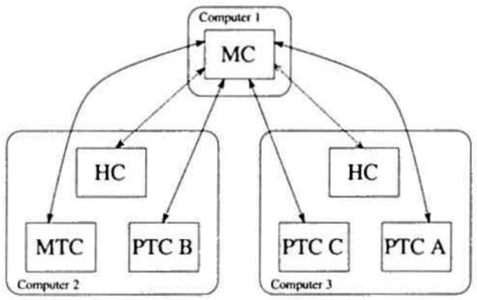

*Fig. 3.* The test system before establishing the port connection

of FSM for this connection and MC has an independent automaton for every port connections in the test system.

Although TTCN-3 allows asymmetric port connections as well, that is, when the list of allowed message types is different in each direction, the arguments of connect and disconnect operations are symmetric. Those operations refer to the same connection even if the arguments are swapped. Therefore the port connections are considered symmetric in our test architecture regarding the tasks for connection setup and termination. The actual tasks, however, are different for the two endpoints. It is the MC's responsibility to choose which end-point shall behave as server or client for the underlying transport connection[4].

### *3.7.1. Connecting TTCN-3 Ports*

The MSC of connection establishment can be seen in *Fig. 4*. First, the test execution on component A reaches the connect statement below:

```
connect(ptcB:port1, ptcC:port1);
```

Because of this, component A sends a connect request message to MC with the component and port identifiers of B and C. Then TC A waits until an acknowledgement message is arrived from MC denoting that the connecting procedure is completely finished.

The connect operation comprises two steps for the MC. Firstly it has to instruct one of the endpoints (component B in our case) to prepare itself for accepting a transport layer connection from the other TC. When B is ready, the MC notifies C to make the connection. When the connection is built up B will send the acknowledgement to MC. This is necessary because the connection establishment is

[4]In the actual MC implementation this selection is based on a canonical ordering of component references and port names.

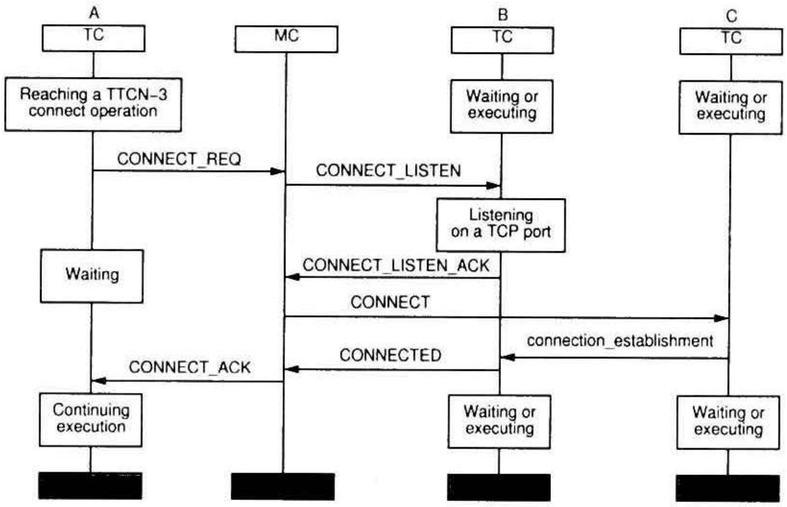

*Fig. 4.* MSC of connect operation

a passive event for the listener side, that is, it may happen that B does not notice the new connection for a while after C became ready.

*Fig. 5* shows the test configuration after the port connection has been established. The thick dashed arrow denotes the new connection.

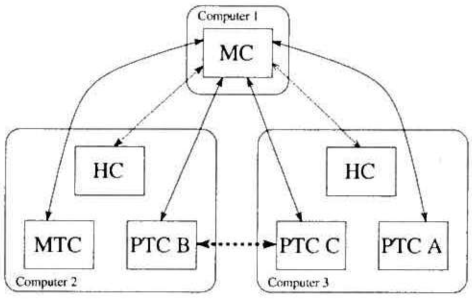

*Fig. 5.* The test system when the port connection is established

### *3.7.2. Disconnecting TTCN-3 Ports*

The communication diagram of a TTCN-3 disconnect operation can be seen in *Fig. 6.* When executing the following disconnect statement:

```
disconnect(ptcB:port1, ptcC:port1);
```

the initiator A behaves very similarly as in the case of connect. It sends a disconnect request to MC and waits until the whole procedure is finished. The task of MC is also easy, but the endpoints have more things to do.

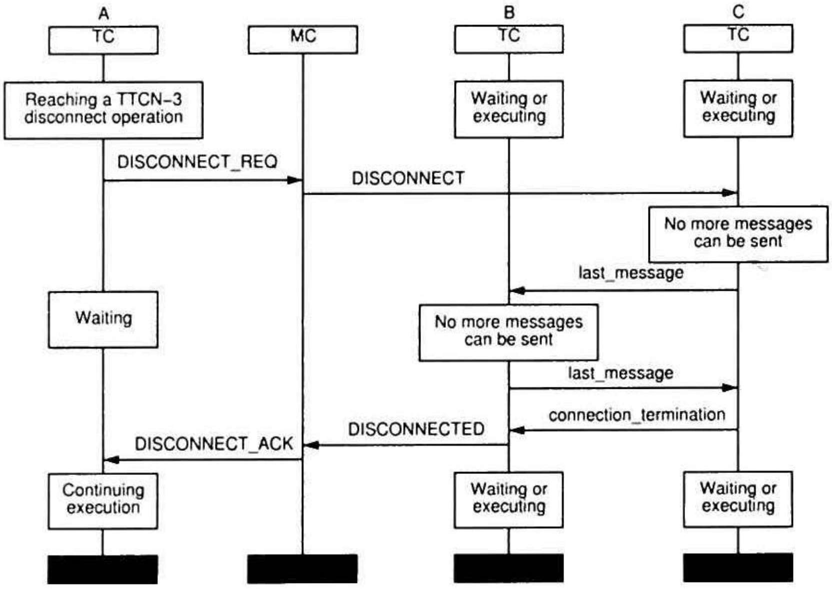

*Fig. 6.* MSC of disconnect operation

The termination of the transport layer connection must be properly arranged in order to avoid message losses in intermediate network buffers. There is a special message called *last_message* that can be transmitted on the transport layer connections of ports. This means that no more messages can be sent after last_message in that direction. The last message cannot be confused with the regular messages of the ports. When MC requests C to disconnect, C sends the last_message to B and waits for another last_message from the opposite direction. When the second last_message has arrived back to C, C can be sure that no messages are staying in the middle of transport connection in either direction. Then C can safely destroy the transport connection, which will be noticed by B and B will send the acknowledgement back to MC.

Finally the test system will get back to its original state as shown in *Fig. 3.*

## 4. Practical Issues

### *4.1. Implementation of Test Architecture*

Our performance test architecture has been successfully implemented as an extension for an existing, compiler-based test executor ([7]), which translates TTCN-3 test specifications into C++ programs.

The processes of Test System were implemented as UNIX processes and the control connections were mapped to simple TCP connections on a local network. The encoding of control messages between the processes was designed to be platform independent so that a group of computers with heterogeneous hardware and operating systems can cooperate and generate load simultaneously.

The key aspects of implementation were the scalability, the robustness and the execution speed. The equivalent C++ code of TTCN-3 test suites can be compiled into efficient executable programs. The test body of typical performance test cases generates a stationary load towards SUT and therefore the number and configuration of PTCs do not change during test run. So the most performance critical part of the Test Architecture, the Main Controller has tasks only during the test set-up and termination phases.

Our real-life experiments showed that this architecture could safely cope with test set-ups with up to 1000 PTCs distributed over more than ten workstations.

### *4.2. Protocol Adaptation*

Like the TTCN-3 language, this test architecture is designed to be independent of the execution platform and the protocol to be tested. However, the handling of port mappings and messages going to or coming from SUT are protocol and platform dependent and outside the scope of the generic test architecture.

To make the test environment flexible, our test executor implementation provides the user with an Application Programming Interface (so-called *Test Port* API) for the protocol specific communication tasks. In the Test Port modules the user has to implement the TTCN-3 `map` and `unmap` port operations in C++ language that shall establish or destroy connections between the Test System and SUT. Sending and receiving messages to or from SUT is also the task of Test Ports, which can be realized with the help of operating system primitives.

The Test Ports are the parts of the processes that realize TTCN-3 test components (either the MTC or a PTC), so the messages of SUT are handled locally. This design principle eliminates the bottleneck of a centralized protocol adaptation.

## 5. Conclusion

Our distributed performance test environment has been successfully applied in a couple of projects. We used TTCN-3 test scripts to generate load against *Remote Authentication Dial In User Service* (RADIUS) and DIAMETER servers. We ran around 500 parallel test components simultaneously distributed on five Sun workstations. Each component emulated one service user by initiating and terminating RADIUS or DIAMETER sessions repeatedly. To generate stationary load, we used random values with given distributions for session duration and session inter-arrival times.

We also used the test environment for the performance evaluation of two experimental IP micro-mobility protocols. In the case of *BRAIN Candidate Mobility Protocol* (BCMP) the existing TTCN-3 conformance test suite was re-used to verify the robustness of system nodes. We simulated 200 end-users that generated both signalling and payload from 2 Linux hosts.

During the investigation of a *Hierarchical Mobile IPv6* (HMIP) system we generated load from two PTCs that ran on a single computer. We measured the delays and latencies of hand-overs. Unfortunately, the only available prototype HMIP implementation was unstable and produced such slow hand-over characteristics that we could not measure repeatable results. Nevertheless, the same test environment can be also used with more settled implementations in the future.

## References

[1] KWIATKOWSKA, M. – NORMAN, G. – SEGALA, R. – SPROSTON, J., *Verifying Quantitative Properties of Continuous Probabilistic Timed Automata*, CONCUR'2000, 2000.

[2] Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3. *The Tree and Tabular Combined Notation*, ISO/IEC 9646-3, 1998.

[3] SCHIEFERDECKER, I. – STEPIEN, B. – RENNOCH, A., PerfTTCN, a TTCN Language Extension for Performance Testing, In *Testing of Communicating Systems*, **10** (1997), Chapman & Hall.

[4] GECSE, R. – KRÉMER, P. – SZABÓ, J. Z., HTTP Performance Evaluation with TTCN, In *Testing of Communicating Systems*, **13** (2000), Kluwer Academic Publishers.

[5] Methods for Testing and Specification (MTS); *The Tree and Tabular Combined Notation Version 3. TTCN-3: Core Language.* ES 201 837-1, ETSI Standard, 2001.

[6] Methods for Testing and Specification (MTS); *Test Synchronization Architectural Reference; Test Synchronization Protocol 1 plus (TSP1+) Specification* ES 201 770, ETSI Standard, 2000.

[7] SZABÓ, J. Z., Experiences of TTCN-3 Test Executor Development, In *Testing of Communicating Systems*, **14** (2002), Kluwer Academic Publishers.