Int J Softw Tools Technol Transfer (2005) 7: 351–360 / Digital Object Identifier (DOI) 10.1007/s10009-004-0165-6

Distributed functional and load tests for Web services

Ina Schieferdecker[1], **George Din**[2], **Dimitrios Apostolidis**[3]

[1] Technical University Berlin/Fraunhofer FOKUS, Berlin, Germany
e-mail: ina@cs.tu-berlin.de
[2] Fraunhofer FOKUS, Berlin, Germany
e-mail: din@fokus.fraunhofer.de
[3] Testing Technologies, Berlin, Germany
e-mail: apostolidis@testingtech.de

Published online: 25 January 2005 –

Abstract. System-level testing considers functionality and load aspects to check how a system performs for single service requests and scales as the number of service requests accessing/using it increases. This paper presents a flexible test framework including functional, service interaction and load tests. It is generic in terms of being largely independent of the system to be tested. The paper discusses the automation of the test framework with the Testing and Test Control Notation TTCN-3 and also presents an implementation of the test framework using a TTCN-3 toolset. The test framework is exemplified for Web service tests and demonstrates distributed functional and load tests for a specific Web service.

Keywords: Web services – Test specification – Distributed tests – TTCN-3 – Test frameworks

Introduction

The Testing and Test Control Notation TTCN-3 has been developed by the European Telecommunication Standards Institute (ETSI) to address testing needs of modern Telco and IT technologies and to widen its scope of applicability. TTCN-3 enables systematic, specification-based testing for various kinds of tests including functional, scalability, load, interoperability, robustness, regression, system, and integration testing. TTCN-3 is a language to define test procedures for black-box testing of distributed systems. It allows an easy and efficient description of complex distributed test behavior 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. TTCN-3 is a modular language that has a similar look and feel to a typical programming language. However, in addition to typical programming constructs, it contains all the important features necessary to specify test procedures and test campaigns like test verdicts, matching mechanisms to compare the reactions of the SUT (system under test) with the expected range of values, timer handling, distributed test components, the ability to specify encoding information, synchronous and asynchronous communication, and monitoring.

This paper discusses the application of TTCN-3 for system-level tests. It describes a test framework with predefined test scenarios and test setups that can be adapted to different systems under test by exchanging the modules for the basic functional tests only. The basic idea is to define a hierarchy of tests for service interaction, scalability, and load tests by reusing basic functional tests for the system under test. Test components are used to emulate system clients. These test components perform the basic functional tests to evaluate the reaction of the system to selected service requests or complex service request scenarios. The combination of test components performing different basic functional tests and being executed in parallel leads to different test scenarios for the systems under test and supports the evaluation of various system aspects. Parameterization of this test framework enables flexible test setups with varying functional and performance load.

The test framework uses a set of basic functional tests for the individual services of a system. Each of these basic functional tests is performed in separate functional tests. A service interaction test checks simultaneous requests of different services by applying several basic functional tests concurrently. A separate load test for individual services checks for scalability and load aspects of the selected service by using several test components with the same basic functional test. A combined load test checks a mixture of requests for different services. These combined load tests use test components performing different functional tests. All the tests return not only a test verdict

but also the response times for the individual requests. A key element of this test framework is its genericity, being largely independent of the concrete system to be tested. In addition, it can be extended to test further aspects.

The application of this test framework to an example Web service is presented. At first, an overview is given of Web services, XML, and SOAP as well as a discussion of testing Web services. Then the test framework is presented. The architecture of the execution environment we used to run the presented test suite is introduced. Selected details of the test framework and its implementation are discussed. Before concluding the paper, measurement results from experiments with load tests are given.

Dinosaur description

```
<schema>
  <element name="dinosaur">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="length" type="string"/>
        <element name="location" type="string"/>
        <element name="place" type="string"/>
        <element name="time" type="string"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Fig. 1. XML schema for the Dino Web service

Web services

A Web service is a URL-addressable resource returning information in response to client requests. Web services are integrated into other applications or Web sites, even though they exist on other servers. So for example, a Web site providing quotes for car insurance could make requests behind the scenes to a Web service to get the estimated value of a particular car model and to another Web service to get the current interest rate.

XML stands for Extensible Markup Language. As its name indicates, the primary purpose of XML was for the marking up of documents. Marking up a document consists of wrapping specific portions of text in tags that convey a meaning, thus making it easier to locate them and also to manipulate a document based on these tags or on their attributes. Attributes are special annotations associated with a tag that can be used to refine a search. An XML document has with its tags and attributes a self-documenting property that has been rapidly considered for a number of applications other than document markup. This is the case with not only configuration files for software but also telecommunications applications for transferring control or application data like, for example, Web pages.

XML follows a precise syntax and allows for checking well-formedness and conformance to a grammar using a Document Type Definition (DTD) that could either be interpreted as a BNF-like grammar specification or in some cases as a data type. A DTD consists of a set of production rules for elements that have a name and describe its content as empty, any, mixed, choice, or sequence. An element can also contain attributes that are declared separately. While DTDs are appropriate for marking up text, they are very limited for other applications because the two basic types CDATA and PCDATA are too general for any precise data typing as in other widely used programming languages. Consequently, the new XML data typing model called Schema was developed. XML schemas [2] are defined using the same basic XML syntax of tags and end tags and actually follow a well-defined DTD. Second, XML schemas are true data types and contain many of the data typing features found in most of the recent high-level programming languages. The central concept of XML schemas is the building-block approach of defining components that consist themselves of type definitions and element declarations. XML schemas are very flexible and allow describing the same rules in many different ways depending on the use of type inheritance, restrictions and extensions, and global and local definitions. Basically, the embedded, flat-catalog and named type structuring approach are distinguished.

This paper uses a dinosaurian database Web service as an example: a dinosaur registration is given as a collection of information about the dinosaur. It is described in terms of name, time, place, length, and location (Fig. 1).

The embedded method derives from the nested tag mechanism of XML itself. In this method, elements are defined where they are used inside the hierarchy. Consequently, there is no need to name a local type – it is called an anonymous type. Eventually the leaves of the tree that constitutes an embedded type definition are composed exclusively of either primitive types or already defined types. This implies that a local definition can be used only once and that there is no need for reusability in a specific application. The flat-catalog approach uses the concept of substitution. Each element is defined by reference to another element declaration. Named types are the closest to traditional computer language data typing. Each element has a name and a type name. Each subtype is defined separately.

SOAP (Simple Object Access Protocol) is a simple mechanism for exchanging structured and typed information between peers in a decentralized distributed environment using XML [4, 5]. SOAP as a new technology to support server-to-server communication competes with other distributed computing technologies including DCOM (Distributed Component Object Model), CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation), and EDI (Electronic Data Interchange). Its advantages are a lightweight implementation, simplicity, open-standards origins, and platform independence.

Testing of Web services

Testing of Web services (as for any other technology or system) is useful for preventing late detection of errors (possibly by dissatisfied users); these typically require complex and costly repairs. Testing enables the evaluation and approval of system qualities and the detection of errors beforehand. An automated test approach helps in particular to efficiently repeat tests whenever needed for new system releases in order to assure the fulfillment of established system features in the new release.

First approaches towards automated testing with proprietary test solutions exist [10]; however, with such tools one is bound to the specific tool and its features and capabilities. Specification-based automated testing, where abstract test specifications independent of the concrete system and independent of the test platform are used, are superior to proprietary techniques: they improve the transparency of the test process, increase the objectiveness of the tests, and make test results comparable. This is mainly due to the fact that abstract test specifications are defined in an unambiguous, standardized notation, which is easier to understand, document, communicate, and discuss. However, we go beyond "classical" approaches towards specification-based automated testing, which till now mainly concentrated on automated test implementation and execution: we consider test-generation aspects as well as the efficient reuse of test cases in a hierarchy of tests.

Testing of Web services has to target three aspects: the discovery of Web services (i.e., Universal Description, Discovery, and Integration, UDDI – which are not considered here), the data format exchanged (i.e., Web Services Description Language, WSDL), and request/response mechanisms (i.e., SOAP). The data format and request/response mechanisms can be tested within one approach: by invoking requests and observing responses with test data representing valid and invalid data formats. Since a Web service is a remote application that will be accessed by multiple users, not only functionality is important in terms of sequences of request/response and performance in terms of response time but also scalability in terms of functionality and performance under load conditions.[1]

The test framework

We have developed a hierarchy of tests for evaluating Web services for functional and load aspects. The basic idea is to define service interaction, scalability, and load tests by reusing basic functional tests for the Web service. Test components are used to emulate Web service clients. These test components perform basic functional tests to evaluate the reaction of the Web service to their requests. The combination of test components performing different basic functional tests in parallel leads to different test scenarios for the Web service. Parameterization of this test framework enables flexible test setups with varying functional and performance load.

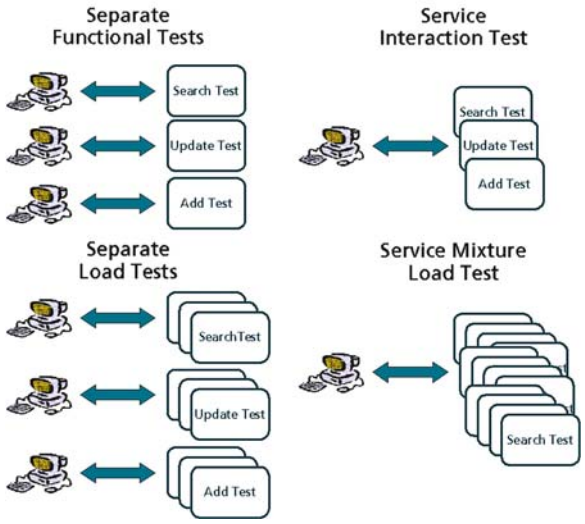

Fig. 2. Test hierarchy for Web services

The basis of the test framework (Fig. 2) is a set of basic functional tests for the individual services of a Web service. It consists of different kinds of tests:

- Separate functional tests for each Web service aspect.
- A service interaction test for simultaneous requests of different services.
- A separate load test for individual service checks for scalability and load aspects.
- A combined load test for a mixture of requests for different services. These combined load tests use test components performing different functional tests.

All the tests return not only a test verdict but also the response times for the individual requests.

An important aspect of this test framework is its genericity of being to a large degree independent of the concrete Web service to be tested. Besides the basic functional tests, fixed test case definitions can be given for the separate functional, service interaction, separate load, and service mixture load. Further test patterns can be envisaged.

The test framework has been realized with the Testing and Test Control Notation TTCN-3 [7], which has been developed by the European Telecommunication Standards Institute ETSI not only for telecommunications but also for software and data communication systems. Like any other communication-based system, Web services are natural candidates for testing using TTCN-3.

Overview of TTCN-3

TTCN-3 is a language to define test procedures for black-box testing of distributed systems and is applicable to

[1] Please note that further test goals for Web services such as availability, resilience etc. exist, which however are not considered here.

both intrusive and nonintrusive testing. Stimuli are given to the SUT; its reactions are observed and compared with the expected ones. On the basis of this comparison, the subsequent test behavior is determined or the test verdict is assigned. If expected and observed responses differ, then a fault has been discovered that is indicated by a *fail* test verdict. A successful test is indicated by a *pass* test verdict.

TTCN-3 allows an easy and efficient description of complex distributed test behavior in terms of sequences, alternatives, loops, and parallel stimuli and responses. Stimuli and responses are exchanged at the interfaces of the system under test, which are defined as a collection of ports. The test system can use a number of test components to perform test procedures in parallel. Like the interfaces of the system under test, the interfaces of the test components are described as ports.

TTCN-3 is a modular language with a similar look and feel to a typical programming language. However, in addition to typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load, and scalability tests like

- Test verdicts,
- Matching mechanisms to compare the reactions of the SUT with the expected range of values,
- Timer handling,
- Distributed test components,
- Ability to specify encoding information,
- Synchronous and asynchronous communication, and
- Monitoring.

A TTCN-3 test specification consists of four main parts:

- Type definitions for test-data structures,
- Template definitions for concrete test data,
- Function and test-case definitions for test behavior,
- Control definitions for the execution of test cases.

The data-type definitions are generated from the corresponding XML schema of the Web service to be tested. The templates are based on the corresponding data types and the behavior of the service being tested, which consist of sequences of requests and responses.

An approach to automated testing of Web services with TTCN-3 therefore requires the following steps (Fig. 3):

1. The structure of the test data is derived from the XML definition, with a set of mapping rules from XML to TTCN-3.
2. Test data (i.e., the concrete values for test stimuli and observations) are generated.
3. Test configuration (i.e., the communication structure between the test system and system under test) is defined to respect the structure of the Web service to be tested.
4. Test behavior (i.e., the sequences of test stimuli and observations) is generated.

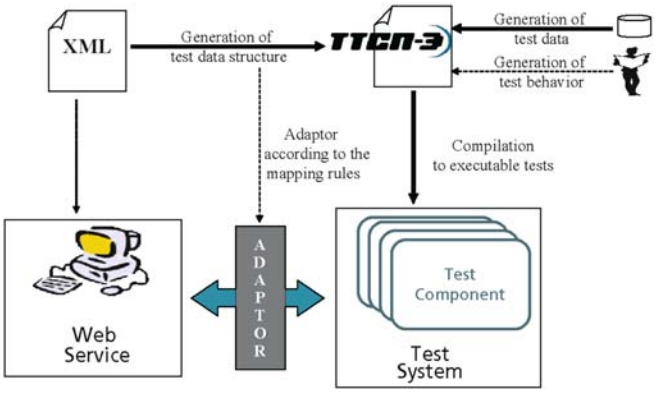

Fig. 3. Testing Web services with TTCN-3

5. The resulting TTCN-3 module is compiled to executable code.
6. The tests are performed using a test adaptor, which follows the mapping rules for test-data structures to encode and decode the Web service requests and replies and to perform the real communication between the Web service under test and the test system.

Currently, steps 1, 4, and 5 can be automated with the help of tools. The automation for steps 2 and 3 requires further work: for this step mainly test-generation approaches based on finite-state machines or labeled transition systems will be used. The test adaptor for step 6 has to be developed only once, so that it can be used for any Web service and TTCN-3 test following the mapping rules from step 1.

The tests

The individual tests in the framework follow the same basic procedure (Fig. 13): the main test component (MTC) creates parallel test components (PTCs) according to the services to be tested (the **create** statement) and according to the load to be generated (the **for** loop). Every PTC gets a concrete test function assigned and is started (the **start** statement). Afterwards, the MTC awaits the termination of all PTCs (the **all component.done** statement). The overall test verdict is the accumulated test verdict of the local test verdicts of the PTCs.

The generic test cases can be controlled with a general test-case control mechanism, shown in Fig. 4. First, in the control part, the functionality of each service offered by a Web service is tested. The results of the tests are recorded and are used as a basis to guide the further execution of the test campaign. If, for example, a functional test for a service fails, it is meaningless to test for service interaction and load aspects for this service. Following the functional tests, load tests for the successfully tested services are performed with an increasing load. Afterwards, service pairs are taken in order to test for service interaction. Finally, the successfully

```
module TestFrameWork {
  type record ServiceLoad {
    integer Service,                                          // the service to be tested
    integer Load                                              // the maximal load for the service
  }
  external const ServiceLoad Services[];                      // array of services to be tested
  external const integer increase;                            // load increase for the load tests
...
  control {
    var integer serviceno:= sizeof(Services);

    var verdicttype ServicesResult[serviceno];                // test result per service

    for (var integer j:=1; j<=serviceno; j:=j+1) {            // functional test per service
      ServicesResult[j]:= execute(SeparateFunctionalTest(Services[j].Service));
    }
    for (var integer j:=1; j<=serviceno; j:=j+1) {            // load test per service
      if (ServicesResult[j] == pass) {
          for (var integer k:= increase; k <= Services[j].Load; j:= j+increase) {
                                                                  // load tests with increasing load
              if (ServicesResult[j] == pass) {
                  ServicesResult[j]:= execute(SeparateLoadTest(Services[j].Service, k));
    } }   }   }


    var verdicttype ServicesMixResult[serviceno][serviceno];  // test result per service pair

    for (var integer j:=1; j<=serviceno; j:=j+1) {            // service interaction test per service pair
      if (ServicesResult[j] == pass) {
          for (var integer k:=1; k<=serviceno; k:=k+1) {
              if (ServicesResult[k] == pass) {
                 const integer ServicePair[2]:= {Services[j].Service, Services[k].Service };
                 ServicesMixResult[j][k]:= execute(ServiceInteractionTest(ServicePair));
    } }   }    }

    for (var integer j:=1; j<=serviceno; j:=j+1) {            // mixture load test per service pair
        for (var integer k:=1; k<=serviceno; k:=k+1) {
           if (ServicesMixResult[j][k] == pass) {
               const integer ServicePair[2]:= {Services[j].Service, Services[k].Service };
               for (var integer l:= increase; l <= Services[j].Load; l:= l+increase) {
                                                                  // load tests with increasing load
                    for (var integer m:= increase; m <= Services[k].Load; m:= m+increase) {
                        const integer PairLoad[2]:= { l, m };
                        ServicesMixResult[j][k]:= execute(MixedServiceLoadTest(ServicePair, PairLoad));
    }   }  }   }    }
  }
}
```

Fig. 4. Execution control for the test framework

tested service pairs are tested for increasing load. The services to be tested, the maximal load for a service test, and the increase for the load tests have to be determined by test execution only – these values are declared as external constants in the TTCN-3 module representing the test framework. The control part can be enhanced to reflect other test combinations, e.g., to cover not only tests for service pairs but also tests for service sets.

Another aspect of this test framework is the evaluation of the final verdict. In functional and conformance testing, every failure detected by a single test component will lead to an overall failure of the complete test. This is also the built-in verdict mechanism of TTCN-3. However, in load tests this is not applicable: a load test checks whether certain thresholds like "99% of requests are successful" are fulfilled. Therefore, a specific verdict type has to be used to handle the collection of the local PTC verdicts and to accumulate them according to the requirements of specific tests.

For that, the MTC was extended to handle the arbitration of PTC verdicts for the overall test verdict.

Realization with TTCN-3

This section completes the discussion of automating the test framework with TTCN-3. A core element of the automation is the definition of an XML to TTCN-3 mapping, which supports the derivation of test-data types from XML schema definitions and is therefore the basis for testing of XML interfaces with TTCN-3. The mapping rules from XML to TTCN-3 have been provided in [13].

Test data

Templates are used to define the concrete test data to be used for requests to and for responses from the Web service. Figure 5 contains example templates to request the "Brachiosaurus" registration and to receive the respective response. The "searchRequest" template makes use of the "requestURL" template, which defines where the information is located and indicates with "Brachiosaurus" the dinosaur to be found. The "Brachiosaurus" response template uses patterns to indicate ranges of acceptable values. For example, the

```
template search searchRequest :=
{
      serviceURL  := requestURL,
      dinoName    := "Brachiosaurus"
}

template url requestURL :=
{
      protocol    := "http://",
      host        := "192.168.89.160:8080",
      webservice  := "dinoserver"
}

template dinosaur Brachiosaurus :=
{
      name        := "Brachiosaurus",
      len         := "22 m",
      mass        := "30 tonnes",
      time        := "Kimmeridgian",
      place       := ?
}
```

Fig. 5. Test data for the Dino Web service

time should be given in the response and must have the concrete value "Kimmeridigian." The location should be given as well, but it could be any place (this is specified by use of the **?** wildcard).

We work on approaches to the automated generation of test data by using the classification tree method [11] being implemented in the CTE tool. This method enables the generation of exhaustive templates for requests. However, it needs to be extended to enable the generation of response templates with patterns as well.

Test configuration

In addition to the structure of the test data, the test configuration in terms of test components and ports has to be generated (Fig. 6). We use a message port to access a Web service. This port can transfer request and response messages. Furthermore, we use a varying set of parallel test components (PTCs) to represent separate functional tests, service interaction tests, separate load tests, and load tests for service mixtures. Every PTC like the SUT has a port to represent the Web service interface.

In the example, the port type "httpTestPortType" defines the capabilities of the Web service to accept "search," "update," and "add" requests and to reply with "dinosaur" information and "updateack" and "addack" acknowledgements. Every PTC is of component type "PTCType" and has one port "httpPort" and a timer "localTimer" to control the timed execution of the test functionality. The interface to the Web service under test is defined by the component type "SUTType" and has one port "httpTestSystemPort".

The PTCs use the same basic test functions to initiate requests and observe responses. The main test component (MTC) controls the dynamic creation of the test components according to the kind of tests. The tests with several components are parameterized, so that the actual numbers of test components emulating the use of a certain service vary depending on the current value of the parameters.

```
type port httpTestPortType message
{
  out         search;
  out         update;
  out         add;
  in          dinosaur;
  in          updateAck;
  in          addAck;
}
type component PTCType
{
  port httpTestPortType   httpPort;
  timer localTimer := 3.0;
}
type component SUTType
{
  port httpTestPortType   httpTestSystemPort;
}
```

Fig. 6. Test components

Basic test function for the Dino Web service

The basic test function for the Dino Web service is depicted in Fig. 7.

It consists mainly of a request and response pair to the Dino service. The search request for Brachiosaurus "searchRequest" is sent and the local timer is started. If the expected response "Brachiosaurus" is received, a pass verdict is assigned. In addition, unexpected and no response are handled – these cases (i.e., the second and third alternative) lead to a fail verdict. The log information logs the received response or the time-out and the respective time stamp.

```
function SeparateSearchFunctional(SUTType sys,
                                  integer service)
runs on PTCType
{
      map(self:httpPort, sys:httpTestSystemPort[service]);
      httpPort.send(searchRequest);
      localTimer.start;

      alt
      {
          [] httpPort.receive(Brachiosaurus)
          {
                  localTimer.stop;
                  log("got expected response");
                  setverdict(pass);
          }
          [] httpPort.receive
          {
                  localTimer.stop;
                  log("unexpected response");
                  setverdict(fail);
          }
          [] localTimer.timeout /*no response*/
          {
                  log("timeout");
                  setverdict(fail);
          }
      }
}
```

Fig. 7. Basic functional test for the Dino service

The map operation at the beginning enables the communication of the PTC with the Dino Web service. This basic test function is specific to the Web service to be tested but has to be developed once and can then be reused for the various types of tests presented above as shown in Fig. 13.

Distributed test platform

The platform previously described to execute the load tests is depicted in Fig. 8. We do not intend to make an exhaustive presentation of all implementation details; rather, we intend to provide an overview of the most important aspects.

The Test Console is the control point of our platform and is an IDE (integrated development environment) that provides support to specify TTCN-3 test cases, to create test sessions, to deploy test suites into containers, and to control the test execution.

Daemons are standalone processes installed on any test device. They manage the containers that belong to different sessions. Containers intercede between the test console and components, providing services transparently to both, including transaction support and resource pooling. From the test viewpoint, containers are the target operational environment and comply with the TCI (TTCN-3 Control Interfaces) standard [16] for a TTCN-3 test execution environment. Within a container, the following specific test system entities exist:

- TE (TTCN-3 Engine) executes the compiled TTCN-3 code. This entity manages different subentities for test control, behavior, components, types, and values and queues entities that realize the TTCN-3 semantics.
- CH (Component Handler) handles the communication between test components. The CH API contains operations to create, start, and stop test components, establish a connection between test components (map, connect), handle the communication operations (send, receive, call, reply), and manage the verdicts. The information about the created components and their physical locations is stored in a repository inside the containers.
- TM (Test Management) manages the test execution. This entity implements operations to execute tests and provide and set module parameters and external constants. The test logging is also tracked by this component.
- CD (Coding/Decoding) encodes and decodes types and values. The TTCN-3 values are encoded into bitstrings that are sent to the SUT. The received data are decoded into abstract TTCN-3 types and values.

The container subentities are functionally bound by the TCI interfaces and communicate with each other via a CORBA [18] platform. The session manager mediation allows many component behaviors to be specified at deployment time rather than in program code. The platform adaptor is a gateway to host resources (i.e., timers). The system adaptor defines a portable service API to adapt to an SUT being defined by the TRI (TTCN-3 Runtime Interface) standard [17]. The test adaptor is a system-specific part and enables flexibility by adapting to the SUT-specific communication and timing. It is typically provided by the test developer.

The whole execution platform is implemented in Java JDK1.4. For the communication between containers we used the CORBA implementation provided with JDK1.4. Following the standard specification helped us to optimize our execution environment for common operations across the test execution. The advantage of adopting a pure Java solution shows in hardware and OS independence, offering the possibility to run tests in various scenarios and deployment patterns.

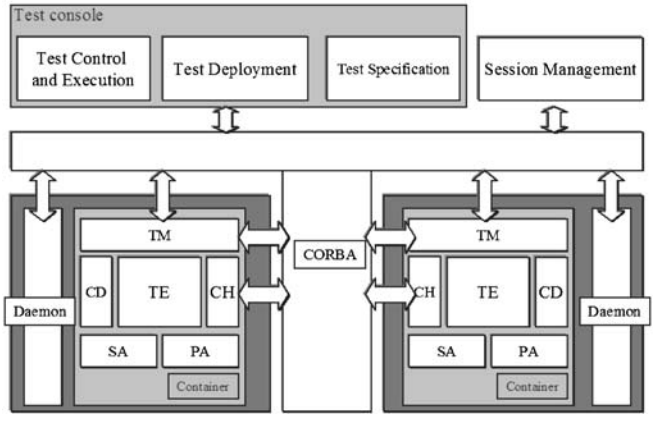

Fig. 8. Distributed test platform architecture

Load test results

We have built a demonstration setup to illustrate the functionality of TTCN-3-based distributed test execution for functional, scalability, and load tests. The architecture of the network we used for our evaluation scenarios is presented in Fig. 9.

We used three standard PCs with 850-MHz CPUs and 256 MB RAM, and another one with a 1.80-GHz CPU and 512 MB RAM. The computers were connected via a 100-Mbps Ethernet hub. We installed on each computer our test containers, which were connected via the CORBA platform. On the last PC we installed our SUT, the Web service. We analyzed the Web service response times under varying load.

We started by distributing a few test components running some functional tests (e.g., SeparateSearchFunctional) and increased the load to up to 56 test components being executed in parallel. We used only equal distribu-

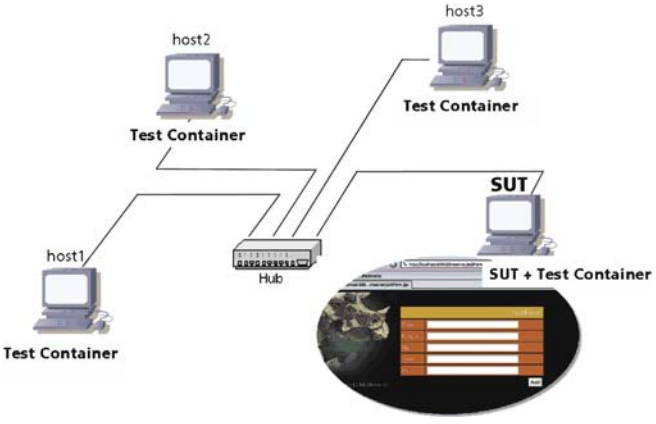

Fig. 9. Test environment setup

tions of components by always running the same number of PTCs on each container. We were not interested in finding out when the SUT would crash but rather to experiment with increasing test load and measuring the changes of the response times in relation to the increasing number of test components.

As the pictures reflect, for very few test components performing requests on the SUT, we measured a very fast reaction of the Web service, under 0.4 ms (Fig. 10). Increasing the number of PTCs (2 per container, 3 per container, and so on) we noticed very quickly an increased latency in the response time. For example, in Fig. 11, when reaching the level of 14 PTCs per test container (which means 56 PTCs at the same time), we measured response times between 0.6 ms and 1.8 ms.

Further, we determined the average response time depending on an increased number of PTCs. We started with distributing 1 component per test container and increased up to 14. Figure 12 reflects this dependency, showing the average response time measured for different test scenarios. For a better view of this dependency, we interpolated the evolution curve.

Conclusion

TTCN-3 is a new test specification and implementation technique (and the only standardized one) that is applicable to a wide range of tests for various system technologies [14]. It is also suited to system-level testing. This

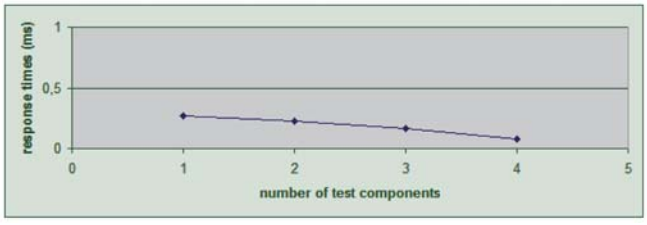

Fig. 10. Four components distributed on four test containers, running functional tests

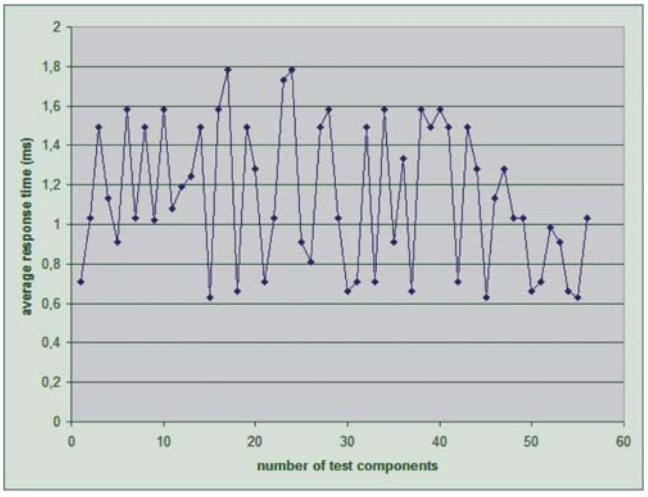

Fig. 11. Response times running 56 test components equally distributed on the 4 test containers

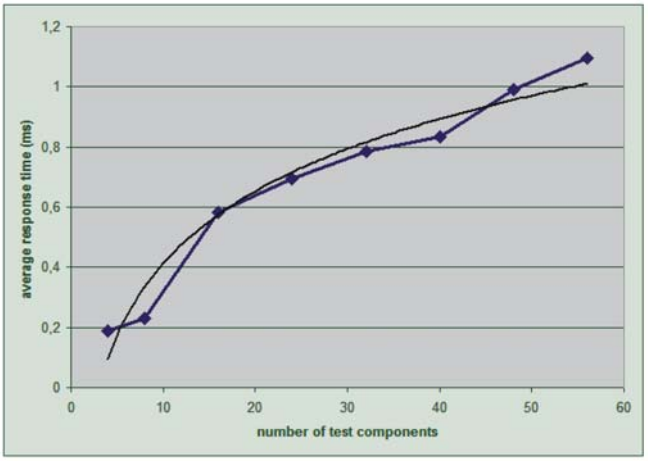

Fig. 12. Average response times for increased load tests

paper discussed system-level tests for Web services with TTCN-3. Beyond the functional and load aspects, further aspects like security, privacy, availability, accuracy, and usability need to be tested and supported by the test framework. This will be addressed in future work.

The paper presented a flexible test framework for Web services realized in TTCN-3. The tool environment supporting this test framework consists of a TTCN-3 to Java compiler TTthree [12], an XML to TTCN-3 conversion tool, and a test adaptor for XML/SOAP interfaces. The adaptor is generic and enables the testing of any Web service using XML/SOAP interfaces. In order to use this adaptor, the mapping rules from XML to TTCN-3 have to be respected by the tests being defined in TTCN-3.

The test framework was developed for Web services with XML/SOAP interfaces and provides functional, service interaction, and load tests with flexible test configurations and varying load. Which aspect of a Web service is tested is defined by basic test functions: a functional test will check for the request/response behavior, a security

```
testcase SeparateSearchFunctionalTest
(integer Service)
runs on MTCType system SUTType
{
  var PTCType PTC:= PTCType.create;
  PTC.start(SeparateSearchFunctional(system,
                                     Service));
  all component.done
}

testcase SeparateLoadTest
(integer Service, integer Load)
runs on MTCType system SUTType
{
  var PTCType PTC[Load];
  for (var integer j:=1; j<= Load; j:= j+1)
  {
    PTC[j]:= PTCType.create;
    PTC[j].start(SeparateSearchFunctional(system,
                                          Service));
  }
  all component.done
}

testcase ServiceInteractionTest
(intarray Service)
runs on MTCType system SUTType
{
  var integer serviceno:= sizeof(Service);
  var PTCType PTC[serviceno];
  for (var integer j:=1; j<= serviceno; j:= j+1)
  {
    PTC[j]:= PTCType.create;
    PTC[j].start(SeparateSearchFunctional(system,
                                          Service[j]));
  }
  all component.done
}

testcase MixedServiceLoadTest
(intarray Service, Load)
runs on MTCType system SUTType
{
  var integer serviceno:= sizeof(Service);
  for (var integer j:=1; j<= serviceno; j:= j+1)
  {
    var PTCType PTC[Load[j]];
    for (var integer k:=1; k<= Load[j]; k:= k+1)
    {
      PTC[k]:= PTCType.create;
      PTC[k].start(SeparateSearchFunctional(system,
                                            Service[j]));
    }
  }
  all component.done
}
```

Fig. 13. Test cases for the different kinds of tests in the test framework

test will check for data integrity, authorization, encryption, etc.

The provided test framework with its test hierarchy is generic as it can be used for arbitrary Web services. The specifics of a concrete Web service are handled within basic test functions emulating the use of the services offered by a Web service. These basic test functions are reused by the kinds of tests provided in the test hierarchy.

A further key element of the test framework is the automated translation of XML data to TTCN-3, so that test skeletons can be generated directly from the specification of a Web service. For that, XML DTDs and schemas have been analyzed and mapping rules have been developed. These rules are realized by a conversion tool from XML to TTCN-3. The conversion tool, together with the TTCN-3 compiler and execution environment TTthree, provides us a complete tool chain for test-data type generation, test development, implementation, and execution.

The principles of the test framework can be applied to other systems and system components such as other middleware or Internet technologies as well. However, if the data specification technique changes, another mapping to TTCN-3 data structures and a corresponding test adaptor will be needed.

While the paper concentrates on functional and load tests, more work is needed on the basic test functions to address additional aspects. Furthermore, test patterns beyond the presented functional, service interaction, and load tests should be investigated. In any case, test automation will be essential to a sound and efficient automated system-level test process for the assessment of the functionality, performance, and scalability of systems.

Future work will further elaborate methods for test-data and test-behavior generation. In particular, the classification tree method will be investigated for potential extension to the generation of TTCN-3 templates. The generation of test-behavior skeletons from sequence diagram specifications is under development. Special emphasis will be given to distributed test configurations with appropriate coordination and synchronization between test components.

The pure TTCN-3 solution for testing of Web services is completed by the implementation of the TTCN-3 TCI standard in our distributed test-execution environment. The implemented platform is manageable, allowing operators to deploy, monitor, and troubleshoot tests as appropriate for the scenario. Future work will address the problem of computing the best strategy to deploy test components to get an optimized performance when executing massive load tests.

The development of the UML 2.0 Testing Profile at OMG [15] will ease the integrated design and development of test systems together with the system itself – system-level tests can be developed at an abstract level on the basis of use cases and use scenarios. The mapping of the UML 2.0 Testing Profile to TTCN-3 enables the direct execution of such tests on TTCN-3 infrastructures.

References

1. W3C (2000) Extensible Markup Language (XML) 1.0. W3C Recommendation, 6 October. http://www.w3.org/TR/2000/REC-xml-20001006
2. W3C (2001) XML Schema Part 0,1,2: Primer, Structures, Datatypes. W3C Recommendations, 2 May. http://www.w3.org/TR/2001/REC-xmlschema-{0,1,2}-20010502
3. Jeliffe R (2000) The XML Schema Specification in Context. http://www.ascc.net/~ricko/XMLSchemaInContext.html
4. W3C (2000) Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May. http://www.w3.org/TR/SOAP
5. McLaughlin B (2002) Java & XML, 2nd edn, Chap 12: SOAP. O'Reilly, Sebastopol, CA

6. Don Box MSDN magazine on the Web (2000) A young person's guide to the simple object access protocol: SOAP increases interoperability across platforms and languages
7. ETSI MTS (2003) The Testing and Test Control Notation TTCN-3, Part 1: TTCN-3 Core Language/ETSI ES 201873-1, October
8. Schieferdecker I, Pietsch S, Vassiliou-Gioles T (2001) Systematic testing of internet protocols – first experiences in using TTCN-3 for SIP. In: 5th IFIP Africom conference on communication systems, Cape Town, South Africa, May 2001
9. Ebner M, Yin A, Li M (2002) Definition and utilisation of OMG IDL to TTCN-3 mapping. In: 16th international IFIP conference on testing communicating systems (TestCom 2002), Berlin, March 2002
10. ANTS (2004) (Advanced .NET Testing System), Red Gate Software. `http://www.red-gate.com/ants.htm`
11. Grochtmann M, Wegener J, Grimm K (1995) Test case design using classification trees and the classification-tree editor CTE. In: Proceedings of the 8th international software quality week, San Francisco, pp 4-A-4/1–11
12. TTthree (2004) (TTCN-3 to Java compiler). Testing Technologies IST GmbH. `http://www.testingtech.de`
13. Schieferdecker I, Stepien B (2003) Automated testing of XML/SOAP based Web Services. In: Proceedings of the GI Fachtagung "Kommunikation in Verteilten Systemen", KIVS 2003, Leipzig, Germany, February 2003
14. Grabowski J, Hogrefe D, Rethy G, Schieferdecker I, Wiles A, Willcock C (2003) An introduction into the Testing and Test Control Notation (TTCN-3). Comput Netw J 42(3): 375–403
15. Schieferdecker I, Dai ZR, Grabowski J, Rennoch A (2003) The UML 2.0 testing profile and its relation to TTCN-3. In: 15th international IFIP conference on testing communicating systems (TestCom 2003), Cannes, France, May 2003
16. ETSI MTS (2003) The Testing and Test Control Notation TTCN-3, Part 5: TTCN-3 Runtime Interfaces/ETSI ES 201873-5
17. ETSI MTS (2003) The Testing and Test Control Notation TTCN-3, Part 6: TTCN-3 Control Interfaces / ETSI ES 201873-6
18. CORBA Technology and the Java 2 Platform (2002) Standard edn, Java 2 SDK, Standard Edition Documentation. `www.java.sun.com`
19. Java 2 Platform (2002) Standard edn, Java 2 SDK. `http://java.sun.com/j2se/1.4/`