2010 IEEE 6th World Congress on Services

# Testbeds for Emulating Dependability Issues of Mobile Web Services

Lukasz Juszczyk, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstraße 8/184-1, A-1040 Vienna, Austria
Email: {juszczyk,dustdar}@infosys.tuwien.ac.at

***Abstract*—Today's ubiquitous internet access has opened new opportunities for mobile workers. By using portable devices, the workers are not only able to access their company's data and/or services from everywhere, but are also offering their own services for being accessible on-demand. The result is on the one hand a higher flexibility, in terms of coordination, but on the other hand poses various challenges to the company's internal workflows due to the dynamic nature of mobility. Consequently, the workflows must be tested at runtime in realistic scenarios in order to get evidence about their correct execution. In this paper we present an approach for emulating mobile workers in order to test the effects of unreliable dependability on workflows. By using the Genesis2 framework we generate testbeds consisting of real Web services and simulate their QoS as well as mobility issues such as packet loss, delay, and an unreliable availability. By generating a running testbed environment, our approach allows to investigate a workflow's execution at to detect runtime faults.**

## I. Introduction and Motivation

In the last years, mobile (or portable) devices, such as netbooks, personal digital assistants (PDAs), and smart phones, have reached a performance level which allows to apply them for more sophisticated purposes, compared to the old days when they served mainly as messengers or calendars. Today, they cover a wide spectrum of applications, from those for pure entertainment and/or simple tasks (as evident in the increasingly popular smart-phone applications) up to critical systems, e.g., for disaster response using mobile ad-hoc networks [1]. This trend has also had a high impact on the domain of mobile workers which perform their tasks on the move, being equipped with portable devices [2]. Initially, their devices served mainly as clients for accessing their company's data and/or services [3], but has evolved towards workers also providing their own Web services [4] which can be called on-demand and integrated into a company's internal workflows [5]. For realizing such systems it makes sense to apply service-oriented computing (SOC) which offers a high level of flexibility due to the decoupling of clients and services. This flexibility is of high importance, since mobile workers are not available 24/7, as it is the case with static software services, but rather provide a volatile availability and, in addition, suffer from unreliable connections. In order to handle these dynamics, the workflows must be able to adapt to unforeseen events in order to guarantee a correct execution. However, we are not dealing with adaptation mechanisms in this paper, but we regard a related problem. How can such workflows be tested with respect to their runtime behavior? Especially for long-running workflow systems, it is necessary to test these in terms of performance, stability, scalability, and to check the adaptation mechanisms in realistic scenarios. Evidently, this calls for testbed infrastructures which emulate such scenarios and allow to verify the workflow's execution before deployment.

In this paper we present a solution which allows to set up such testbeds in a convenient yet flexible manner. Our work is based on the Genesis2 framework [6] (in short, G2) and supports the creation of running testbeds out of script-based specifications. In a nutshell, engineers are free to specify the structure and behavior of the testbed, consisting for instance of Web services, clients, registries, and to generate running instances of these at a distributed back-end. We have extended G2 in order to emulate dependability issues of mobile workers and to control these via the Groovy [7] scripting language.

We present our approach as follows. In the current section we outlined a short motivation for our research. Section II, is the main part of the paper, and describes our contribution, its benefits, and limitations. Finally, in Sections III and IV we compare our approach to related work and conclude this paper.

## II. Our Approach: Genesis2 & Traffic Control

The aim of our tool-based solution is to support engineers in generating testbeds which emulate an environment of mobile workers, including the inherent dynamics which pose a challenge to workflow systems. These dynamics include volatile availability, network delays, packet losses, and quality of service (QoS) in general. Our approach combines the G2 framework, which handles the generation of SOA testbeds, with the Linux tool *traffic control* in order to inject communication faults for simulating dependability issues.

### A. Genesis2 Testbed Generator

In the last years we realized a gap between the progress of research on service-oriented computing (SOC) in general and on testing solutions for SOC. So far, work has been mainly done on testing single Web services, but it has been neglected to work on solutions for testing large systems which operate on service-based environments themselves, such as active registries, governance systems, or workflow engines. Runtime-based tests for these systems require testbed environments, consisting of Web services, clients, service buses, mediators, etc. However, the set up of such testbeds has

978-0-7695-4129-7/10 $26.00 © 2010 IEEE
DOI 10.1109/SERVICES.2010.83

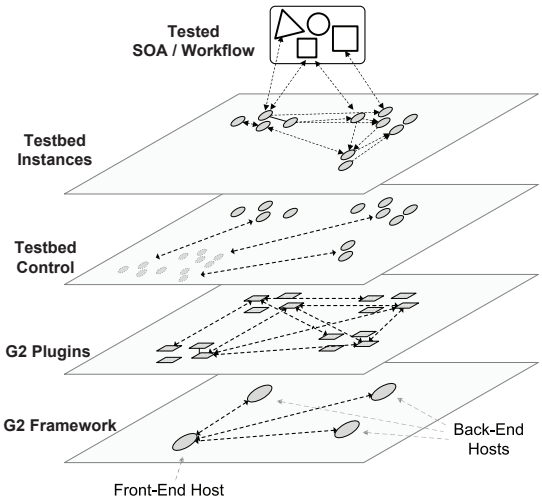


Fig. 1. Layers of interaction within a Genesis2-based testbed

been a cumbersome and time consuming task due to the lack of proper tool support. In order to close this gap, we have developed the G2 framework which aims at providing means for engineers to specify testbeds, program their behavior, and generate automatically running instances of these. Due to space constraints it is not possible to present all details of G2. Instead, we refer interested readers to [6] and summarize the most relevant features in this paper.

G2 comprises a centralized front-end, from where testbeds are modeled and controlled (via Groovy scripts), and a distributed back-end at which the models are transformed into real testbed instances. The front-end maintains a virtual view on the testbed, allows engineers to manipulate it on-the-fly via scripts, and propagates changes to the back-end in order to adapt the running testbed. For the sake of extensibility, G2 uses composable plugins which augment the testbed's functionality, making it possible to emulate diverse topologies, functional and non-functional properties, and behavior. Figure 1 depicts a simplified view on the different layers of a G2-based testbed and the interactions within them.

- At the lowest level, G2 maintains connections between the front-end and the back-end hosts in order to synchronize the testbed model and to propagate plugins.
- On top of this, the individual plugins are deployed, which are free to implement their own communication strategies, e.g., for data exchange via gossiping.
- Based on the G2 framework and the functionality provided by the plugins, engineers create models of their testbeds and deploy these on the back-end where G2 takes care of generating real instances. At the front-end, the models comprise virtual objects which represent the generated instances and act as proxies for on-the-fly manipulations.
- The generated testbed consists of the elements modeled by the engineer and behaves according to its specification, e.g., by having nested service invocations, registry queries, etc. All in all, the testbed's behavior is fully customizable and not restricted by the G2 framework.
- Eventually, on top of this emulated environment, the workflow or service-oriented architecture (SOA) can be tested.

In summary, the engineer specifies the testbed at the front-end, defining *what* shall be generated *where*, with *which customizations*, and the framework takes care of synchronizing the model with the corresponding back-end hosts on which the testbed elements are generated and deployed. Listing 1 contains a sample script for modeling a Web service, programming it's behavior (in this case just returning a String value), and deploying it at a back-end host. After deployment, it is possible to perform adaptations on-the-fly by changing the Web service's model which is immediately propagated to the generated instance at the back-end.

```
def dt=datatype.create("/schemas/types.xsd","vCard")

def service = webservice.build {
  // create model of TestService with one operation
   TestService(binding: "doc,lit", tags: ["test"]) {
      SayHello(card: vCard, result: String) {
         def name = card.name
         return "hello $name"
      }
   }
}[0]

// import back-end host reference
def beHost1 = host.create("192.168.1.11:8080")

service.deployAt(beHost1) // deployment at back-end

service.operations += ... // on-the-fly adaptations
```

Listing 1. 'Specification and deployment of a Web service'

For the purpose of emulating mobile worker's Web services, we are using the G2 framework for generating the basic testbed. Furthermore, we apply plugins for simulating QoS and dependability issues of mobile workers.

### B. Emulating Mobile Web Services

Web services on portable devices suffer from two kinds of problems: unreliable connectivity, caused by the nature of wireless communication, and an unsteady availability of the human worker. If such services need to be incorporated into a company's workflow or service-oriented architecture, these systems must be able to handle the dynamics inherent in mobile computing and must be tested in simulated scenarios. Consequently, they require a testbed which emulates mobile workers and their dependability problems.

In our approach we make use of the distributed nature of a G2 testbed and assign each worker a separate back-end host and deploy a customizable set of services which represent the workers' repertoires. The emulation of connectivity problems and of QoS takes place on two different levels. While connectivity problems usually affect whole devices, including the access to all deployed services, QoS properties are local to

individual services, or to be more precise, to their operations. To achieve an effective emulation of these, we have extended G2 with two additional plugins: a *network emulator*, for low level fault injection, and a *QoS plugin*, for emulating QoS properties such as processing time, throughput, and scalability.

*1) Network Emulator Plugin:* For emulating network faults we are using the Linux tool *traffic control* (`tc`) [8] in combination with the *netem* module [9]. Basically, for each back-end host, which represents a worker's device, the plugin creates a virtual IP address which can be then controlled via `tc`:

```
ifconfig lo:0 add 192.168.1.11
```

In order to control the fault behavior via the front-end, the plugin defines extensions to the model of back-end hosts, so that the testbed used can steer a host's properties via simple variable assignments:

```
beHost1 { // manipulate host properties
   tc.loss = 0.025 // packet loss
   tc.delay = 1500 // packet delay
   tc.corrupt = 0.001 // packet corruption
}
```

G2 intercepts these manipulation request and propagates the changes to the corresponding back-end hosts at which they are translated into `tc` commands:

```
tc qdisc change dev lo:0 root netem loss 2.5%
tc qdisc change dev lo:0 root netem delay 1500ms
tc qdisc change dev lo:0 root netem corrupt 0.1%
```

As a result, the network emulator plugin makes it possible to deploy back-end hosts on virtual IP addresses and to control their emulated communication properties from remote.

*2) QoS Emulator Plugin:* For emulating quality of service, we have ported the *QoSPlugin* of the first version of Genesis [10] to G2. In contrast to the network emulator which augments the hosts, the *QoSPlugin* attaches itself to the Web services and their operations in order to make them behave according to specified non-functional attributes. Currently, the spectrum of supported QoS attributes includes processing time, invocation throughput, scalability on parallel invocations, service availability, and accuracy [11]. Again, the plugin is being controlled via variables and simulates QoS by delaying service invocations, throwing exceptions, and altering the service's deployment status:

```
service {
   // manipulate whole service's properties
   qos.processingtime = 60 // 1 minute
   qos.throughput = 5/60 // 5 tasks per hour
   qos.availability = 0.98 // 98% availability

   // manipulate QoS of single WS operation
   SayHello.qos.throughput = 10/60
}
```

### C. Illustrating Example

In the following we are using a sample specification script for demonstrating the practical application of our approach. Due to the limited space in this paper, the script manipulates the attributes for emulation of network failures and QoS only by assigning randomized values. This behavior does obviously not emulate a realistic scenario, which would rather require more sophisticated strategies for distribution of workers and simulation of their behavior. However, for demonstration purposes it shows how the testbed can be steered.

The script starts with referencing 20 back-end hosts at virtual IPs and importing message type definitions from XSD files. In Lines 7-23 a basic set of worker's Web services is defined which are then all deployed on every single host (Lines 25-29). Finally, in Lines 31-43 a background thread is started for each host, which controls the emulation of network connectivity and QoS by changing the corresponding variables.

```
11.upto(30) { n-> host.create("192.168.1.$n:8080") }

def stat=datatype.create("/my/types.xsd","typeName")
def task= ... // import message data types from XSD

def serviceSet=webservice.build {
  // define required Web services & operations
  StatusService(binding: "rpc,enc") {
    GetWorkerStatus(response:stat) {
      return ... // current worker status
    }
  }

  TaskManagerService() {
    AssignTask(t : task, response: String) {
      // ... program operation behavior
    }
    StopTask(taskID: String, response: boolean) {
      // ... program operation behavior
    }
  }

  // more services
}

host.getAll().each { h -> // on each host ...
  serviceSet.each { s -> // deploy all services
    s.deployAt(h)
  }
}

host.getAll().each { h ->
  Thread.start {
    while (running) { // boolean flag
      Thread.delay(5000) // in 5sec intervals
      // set randomized attributes
      h.tc.loss = new Random().nextFloat()
      // for each deployed service
      h.webservice.getAll().each { s ->
        s.qos.availability= ... // random attribs
      }
    }
  }
}
```

Listing 2. 'Simplified specification of testbed emulating mobile workers'

For the sake of brevity, we have restricted the demonstrated testbed to only emulating mobile Web services. In reality, depending on the requirements of the tested system, a proper testbed would also incorporate registries, client functionality for invoking services, message interceptors, etc. Details, about how such a testbed can be specified and generated, are explained in our previous work [6].

### D. Limitations

Of course, our approach is not capable of emulating all aspects of mobile computing. All it does is generating Web services and, if desired, also other SOA components, and binding them to hosts which are controllable remotely. In other words, it emulates a dynamic environment (= creates a mock-up) for testing of a workflow or SOA operating on top of that environment. Our approach does not aim at emulating the runtime of mobile devices in order to test software on these. For this, engineers should use device emulators, e.g., from Microsoft [12] or Sun Microsystems [13].

Furthermore, currently we are only able to simulate simple QoS of workers, which do not provide a high level of realism. We regard it necessary to emulate the workers' behavior also regarding working times, variable working speed, and other relevant attributes of human work. This will be addressed in future work.

## III. Related Work

In the domain of testbed generation, the projects/prototypes of SOABench [14] and PUPPET [15] provide a functionality similar to that of G2. SOABench aims at benchmarking BPEL workflow engines [16] via modeling experiments and generating service-based testbeds. It provides runtime control on test executions as well as mechanisms for test result evaluation. Regarding its features, SOABench is focused on performance evaluation and generates Web service stubs that emulate QoS properties, such as response time and throughput. Similar to SOABench, PUPPET generates QoS-enriched testbeds for verifying service compositions, but is more focused on the verification of Service Level Agreement (SLA) fulfillments for composite services. Similar to G2, these two approaches serve their purposes by generating service-based testbeds which emulate QoS. Yet, G2 is not restricted to a specific domain but is highly customizable via plugins which introduce aspects of emulated environments, such as our network emulator which injects communication faults. Moreover, G2 allows to program the behavior of generated Web services in order to customize it to the testing purpose.

For the emulation of communication faults, we are using *traffic control* [8] and *netem* [9] which provide all necessary functionality to inject faults on a single host. Related to this, we could have also used network emulators such as ns2 [17] or GloMoSim [18], which provide even more sophisticated functionality than *netem* but at the cost of sacrificing simplicity. Furthermore, emulators for mobile ad-hoc networks, such as Octopus [19], MobiNet [20], or MobiEmu [21] do exist, of which each has different strengths and weaknesses. Also in this case, it would be possible to integrate our approach with one of these emulators. All in all, the purpose of our work has never been to compete with these tools but to integrate them with G2 for emulating mobile Web service-based computing.

## IV. Conclusion

In this paper we have presented our approach for emulating mobile workers' Web services in order to test the effects of a volatile dependability on a service-oriented architecture at runtime. By using the Genesis2 framework, we are able to generate testbed environments which we augment with plugins for simulating mobility issues, such as network failures as well as quality of service attributes. The strengths of our approach are the high degree of customizability, which allows to generate testbeds of arbitrary structure and behavior, and the convenience of using a compact scripting language for modeling the testbeds.

## Acknowledgment

The research leading to these results has received funding from the European Community Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

## References

[1] T. Catarci, M. de Leoni, A. Marrella, M. Mecella, B. Salvatore, G. Vetere, S. Dustdar, L. Juszczyk, A. Manzoor, and H. L. Truong, "Pervasive software environments for supporting disaster responses," *IEEE Internet Computing*, vol. 12, no. 1, pp. 26–37, 2008.

[2] M. Perry, K. O'Hara, A. Sellen, B. A. T. Brown, and R. H. R. Harper, "Dealing with mobility: understanding access anytime, anywhere," *ACM Trans. Comput.-Hum. Interact.*, vol. 8, no. 4, pp. 323–347, 2001.

[3] M. Chen, D. Zhang, and L. Zhou, "Providing web services to mobile users: the architecture design of an m-service portal," *IJMC*, vol. 3, no. 1, pp. 1–18, 2005.

[4] D. Schall, R. Gombotz, C. Dorn, and S. Dustdar, "Human interactions in dynamic environments through mobile web services," in *ICWS*. IEEE Computer Society, 2007, pp. 912–919.

[5] D. Schall, H. L. Truong, and S. Dustdar, "The human-provided services framework," in *CEC/EEE*. IEEE, 2008, pp. 149–156.

[6] L. Juszczyk and S. Dustdar, "Script-based generation of dynamic testbeds for soa," in *ICWS*. IEEE Computer Society, 2010.

[7] "Groovy Programming Language," http://groovy.codehaus.org.

[8] "Linux Advanced Routing & Traffic Control," http://lartc.org.

[9] "netem - Network Emulator," http://www.linuxfoundation.org/en/Net:Netem.

[10] L. Juszczyk, H. L. Truong, and S. Dustdar, "Genesis - a framework for automatic generation and steering of testbeds of complexweb services," in *ICECCS*. IEEE Computer Society, 2008, pp. 131–140.

[11] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping performance and dependability attributes of web services," in *ICWS*. IEEE Computer Society, 2006, pp. 205–212.

[12] "Microsoft Device Emulator," http://go.microsoft.com/fwlink/?LinkId=71203.

[13] "Java ME Emulator Toolkits," http://java.sun.com/javame/sdk/.

[14] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware," Faculty of Informatics - University of Lugano, Tech. Rep. 2009/07, November 2009. [Online]. Available: http://www.inf.usi.ch/research_publication.htm?id=55

[15] A. Bertolino, G. D. Angelis, and A. Polini, "A qos test-bed generator for web services," in *ICWE*, ser. Lecture Notes in Computer Science, vol. 4607. Springer, 2007, pp. 17–31.

[16] "OASIS - Business Process Execution Language for Web Services," http://www.oasis-open.org/committees/wsbpel/.

[17] "The Network Simulator - ns2," http://www.isi.edu/nsnam/ns/.

[18] "Global Mobile Information System Simulator," http://pcl.cs.ucla.edu/projects/glomosim/.

[19] F. D'Aprano, M. de Leoni, and M. Mecella, "Emulating mobile ad-hoc networks of hand-held devices: the octopus virtual environment," in *MobiEval*. ACM, 2007, pp. 35–40.

[20] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat, "Mobinet: a scalable emulation infrastructure for ad hoc and wireless networks," *Mobile Computing and Communications Review*, vol. 10, no. 2, pp. 26–37, 2006.

[21] Y. Zhang and W. Li, "An integrated environment for testing mobile ad-hoc networks," in *MobiHoc*. ACM, 2002, pp. 104–111.