The Test Automation Platform as a Service

 © 2017 – All rights reserved

Table of contents

SIMPLY TESTIFY

- The testing problem
- What is Test Automation?
- Test automation 2.0
- The simplyTestify solution
- Automation of cognitive test tasks
- Model/policy based test automation
- Low code test automation
- Contract-based test automation
- Automated testing capability
- Automatic test arbitration
- Test automation as a service
- simplyTestify core architecture
- Conclusion

Annexes

- Unit test of a terminal component
- Unit test of a non-terminal component
- Integration test of a distributed SUT
- Integration test of a distributed SUT (II)

 © 2017 – All rights reserved

The testing problem

Always-on world - even non mission-critical IT system become **business-critical**

Failures in the production stage have strongly adverse business consequences: **catastrophic outcomes**, **physical injuries**, **regulation infringements**, **reputational damages**, **liabilities**, **customer churn**, **profit losses**, **competitive disadvantages**, **time-to-market delays**, and **extra labor and equipment costs**

Nowadays, **systematic and effective test** in several points of the **DevOps process** becomes **mandatory**

Manual test is **hard**, **knowledge-intensive**, **time-consuming**, very **expensive in labor and equipment**, and often **ineffective**, **inefficient**, and **error-prone**

Test automation is the **solution**

© 2017 – All rights reserved

What is Test Automation?

Test Automation is a buzzword: tools currently available in the QA & Testing market **implement at most the bare mechanization of clerical test tasks**

Agile, TDD and BDD methodologies and **Continuous Integration and Delivery DevOps** processes **need higher levels of test automation**

Leading organizations develop **complex, custom-built, tailor-made, and ad hoc test systems**, carried out by **challenging and expensive software projects** borne by **highly skilled developers**

Resulting home-made test systems are not only very limited in automation (only the clerical test tasks are mechanized) but also so **burdensome, brittle, error-prone, and dear to maintain, configure, deploy, and operate** – often more than the system to be tested - that many customers revert to **manual, outsourced, and offshore testing**

Anyway, **high code test automation is out of range of small and medium organizations**

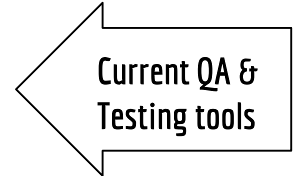
 © 2017 – All rights reserved

Test automation 2.0

Customer challenges generate needs that are not satisfied by the currently available technology and can be met only by full Test Automation

Automation levels

	Test generation	***Test run***	***Test management***
2	Autonomous, intelligence-led, robotics, and low code automated design and generation of synthetic, context aware, and focused test scenarios and samples (inputs and oracles) from models and policies	Autonomous, intelligence-led, robotics, and low code automation of the test runs, with automated configuration and binding, dynamic intelligent scheduling, on the fly focused generation, and evidence-based reactive planning	Fault-tolerant API/scripting invocation of autonomous test automation software robots running on a highly elastic, dependable, secure and performant platform
1	Mechanized generation of test input and oracle templates. User manual drafting of test inputs and oracles	High code configuration and binding of custom-built test systems. Mechanization of batches of sequential test execution. Eye-ball arbitration of SUT responses by inspection of bulky journals. Manual reporting	DevOps process invoking custom-built and unstable test systems without any guarantee of elasticity, dependability and performance
0	Manual drafting of test inputs and oracles	Manual configuration and binding of a client software. Manual sending of the test inputs and gathering the SUT responses. Eye-ball arbitration and manual reporting	Human workflow for an activity that uses disputed equipment and is carried out by scarce human resources, upon software of uncertain quality



© 2017 – All rights reserved

The SIMPLYTESTIFY solution

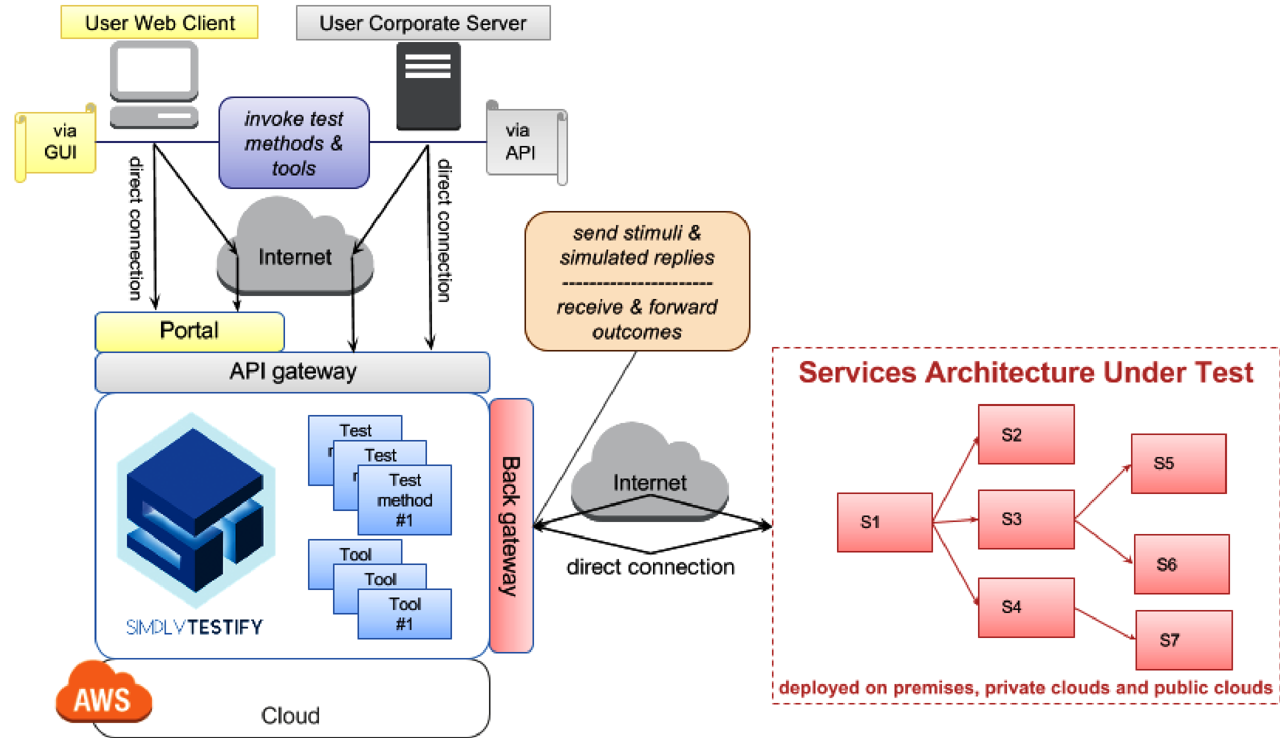
The use of SIMPLYTESTIFY is straightforward, declarative and low-code

The user

1. deploys the SUT in the distributed testbed (on-premises, on private, public, and hybrid cloud);
2. drafts a few declarative artifacts (models and policies);
3. implements the SUT Initialization API

The SIMPLYTESTIFY robots

- autonomously design and generate synthetic test scenarios and data sets,
- run dynamically scheduled test sessions,
- produce and summarize test verdicts, and
- plan and implement multi-session test campaigns.



SIMPLYTESTIFY

 © 2017 – All rights reserved

Automation of cognitive test tasks

The tools currently available in the QA & Testing market do not provide any support for the automation of **cognitive test tasks**

SIMPLYTESTIFY robots automate:

- the **derivation of test scenarios** from the **models** and **policies**
- the **design of test inputs** with elevated **fault exposing potential** from the **models** and **policies**
- the **derivation of the test oracles** from the **SUT models**
- the **generation** of **executable test samples** (scenarios' instances)
- the **configuration** of the **test harness** with **simulated components** (**clients**, **stubs**) and **intercepting proxies**
- the **binding** of the **test harness** to the **SUT** on the **distributed testbed**
- the **arbitration** of the **SUT responses**
- the **dynamic prioritization** of the **test samples**, based on probabilistic reasoning and driven by failure search policies and past test verdicts
- the **evidence-based planning** of **test campaigns**, based on probabilistic reasoning and driven by coverage policies, with on-the-fly generation of new test scenarios/samples
- the **reporting** of the **test sessions**

 © 2017 – All rights reserved

Model/policy based test automation

SIMPLY TESTIFY

The user shall supply **the native interface and protocol descriptions** (e.g. WSDL for SOAP, WADL or WSDL for REST/XML, OpenAPI/Swagger for REST/JSON …)

SIMPLYTESTIFY test automation is based on a few additional **declarative artifacts** (XML format):

Models

- **SUT topology** - graph of SUT components linked by service dependency wires
- **Test harness configuration plan** - structural and behavioral model of the simulated upstream and downstream components (respectively, ***clients*** and ***stubs***) and of the intercepting ***proxies*** on the service dependency wires
- **Protocol state machines** - for each actual (SUT) and virtual (Test harness) component, a state/transition (event/condition/effect with transfer functions) representation of the allowed conversations with its upstream ad downstream services

Policies – **generation**, **scheduling**, and **planning** policies

© 2017 – All rights reserved

Low code test automation

SIMPLYTESTIFY

Low code automation: the only coding effort requested from the user concerns the (re)initialization procedure for each SUT component

- Each (re)initialization procedure implements the **SUT initialization APIs**
- The **executor/arbiter robot automatically invokes the initialization APIs** on each SUT component: (i) at the start of the test session, and (ii) at the end of each test run
- **SIMPLYTESTIFY** ensures the **repeatability of the test runs** by implementing the ETSI* recommendation that each test run must leave the SUT in its initial state
- The **setup of the test system** does not require **any specific coding effort**

* European Telecommunication Standard Institute - http://www.etsi.org/

 © 2017 – All rights reserved

Contract-based test automation

SIMPLY TESTIFY

The user can model **pre/post-conditions** on **stateful SUT components**

She shall

- define a **state resource** (XML infoset) that represents the **externally visible state** of the component
- implement a the **Contract-Based Test API** on the component for **setting/getting the state resource**

The executor/arbiter robot **automatically invokes these APIs** on the SUT components at particular moments of the test run

- to **set** the **initial component state** for the test run
- to **get** the **actual component state** for checking whether it meets the **pre/post-conditions**

 © 2017 – All rights reserved

Automated testing capability

SIMPLYTESTIFY

The **SIMPLYTESTIFY executor/arbiter** can handle:

- **protocol testing** – of the compliance of the actual message with the oracle message type,
- **content testing** – of the compliance of the actual message content with the oracle message content,
- **contract testing** – of the compliance of the actual SUT state with the oracle SUT state,
- **fault-tolerance testing** – of the component behavior in the presence of the failure of downstream services (stubs that do not respond),
- **non-determinist test scenarios** - where asynchronous SUT feedbacks are allowed arriving in an unspecified order.

N.B.: the features mentioned above are applied to **gray-box testing of complex and large-scale distributed architectures**.

 © 2017 – All rights reserved

Automatic test arbitration

SIMPLYTESTIFY

SIMPLYTESTIFY automates the **arbitration task**

For each test run, the executor/arbiter initializes a **compound test verdict**, i.e. an **array of local verdicts** (one for each **expected SUT feedback** in the **test scenario – actual message** or **retrieved state**) - all the local verdicts are initialized to ***none***

The **test components** (**clients**, **stubs**, **proxies**) set the **local verdicts** for the **SUT feedbacks** they arbitrate to

- ***pass*** - the **SUT feedback matches** the **oracle**
- ***fail*** - the **SUT feedback mismatches** the **oracle**

The occurrence of a ***fail*** local verdict or a ***timeout*** ends the test run. The verdicts of the non received SUT feedbacks remain unchanged (***none***)

At the end of the test run, the executor/arbiter re-initializes the SUT and returns the compound test verdict

© 2017 – All rights reserved

Test automation as a service

SIMPLYTESTIFY implements

- the **reliable invocation** of **test automation methods** via APIs
- the **reliable execution** of **test automation methods** by **autonomous robots** on a highly elastic, dependable, and secure cloud infrastructure

SIMPLYTESTIFY enables the **swift wrapping** of **test automation services** in **all types of DevOps automated processes**:

- **automated black-box test of components** starting at the early stages of design and development (**shift left testing**);
- **automated black-box test of partners' and third-parties' components** (**client-driven testing**)
- **automated gray-box test** throughout all the process of **incremental integration** of distributed systems (**incremental testing**)
- **automated regression test** of new and updated **component builds** (**acceptance testing**)
- **on-the-fly automated test** of **suspicious components** triggered by the ops monitoring system (**shift right testing**)

Continuous Integration

Continuous Delivery

Continuous Development

Continuous Testing

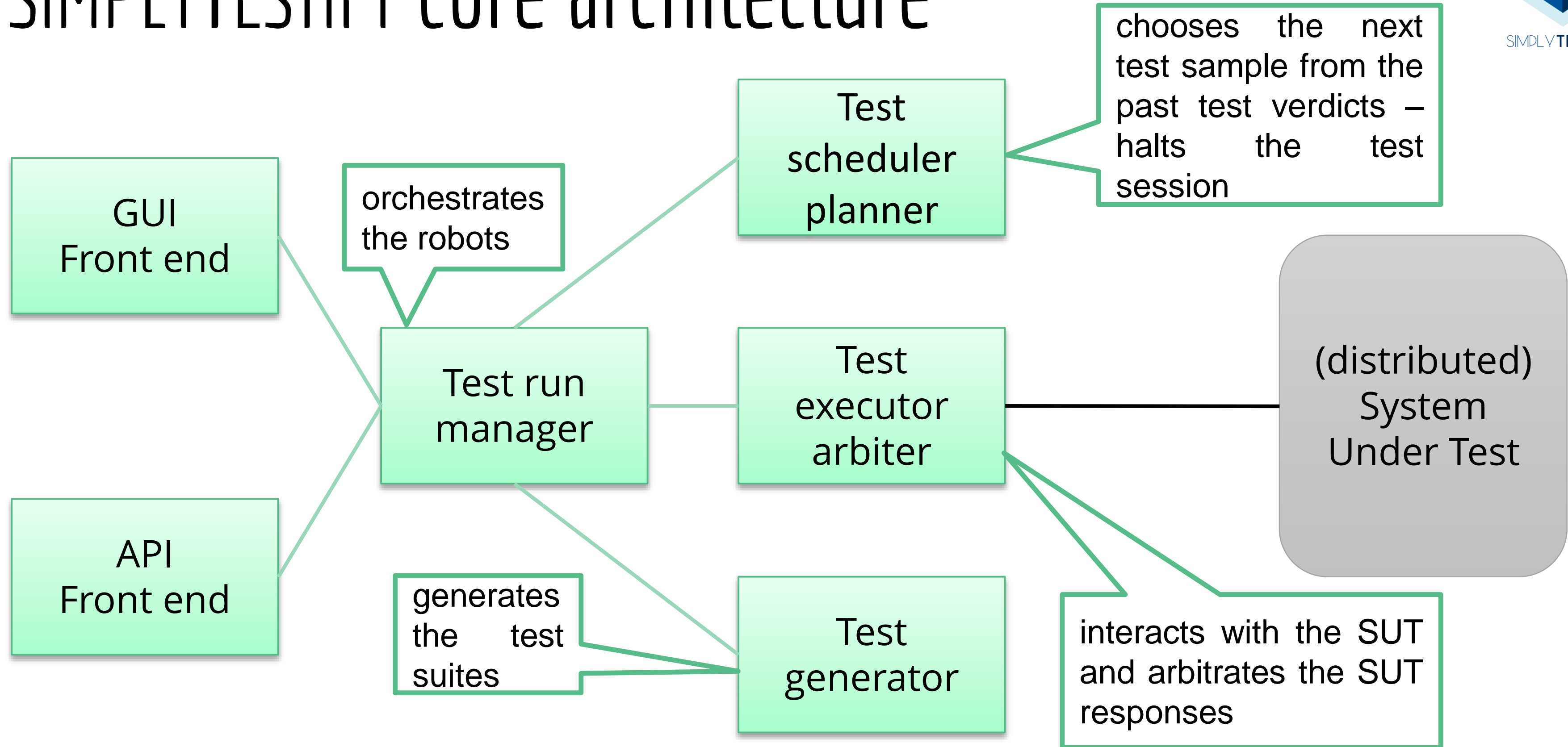
Continuous Deployment

Change Management

Ops Monitoring

SimplEngineering © 2017 – All rights reserved

SIMPLYTESTIFY core architecture



SimplEngineering © 2017 – All rights reserved

Conclusion

SIMPLY TESTIFY

Test Automation: full, autonomous, intelligence-led, robotics, low-code, as a Service

Black-box, contract-based, client-driven Test Automation

Gray-box Test Automation of complex and large-scale distributed system

Availability

- Platform as a Service on public cloud – self-provisioning, pay-as-you-go, low-cost – today AWS
- Can be licensed on private cloud

© 2017 – All rights reserved

Thanks

Libero MAESANO
libero.maesano@simple-eng.com

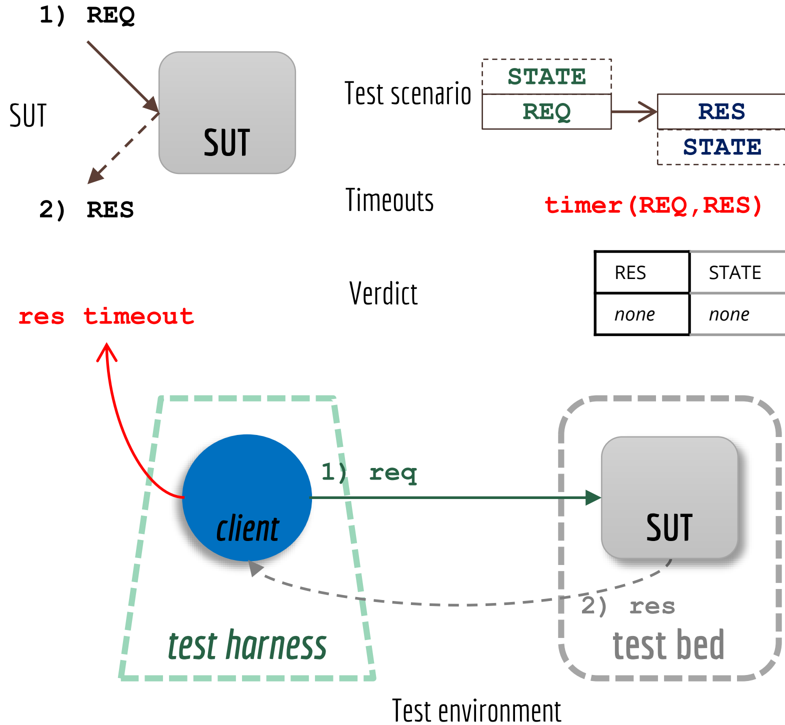
 © 2017 – All rights reserved

Annexes

- Unit test of a terminal component
- Unit test of a non-terminal component
- Integration test of a distributed SUT
- Integration test of a distributed SUT (II)

SimplEngineering © 2017 – All rights reserved

Unit test of a terminal component



(*The SUT is initialized*)

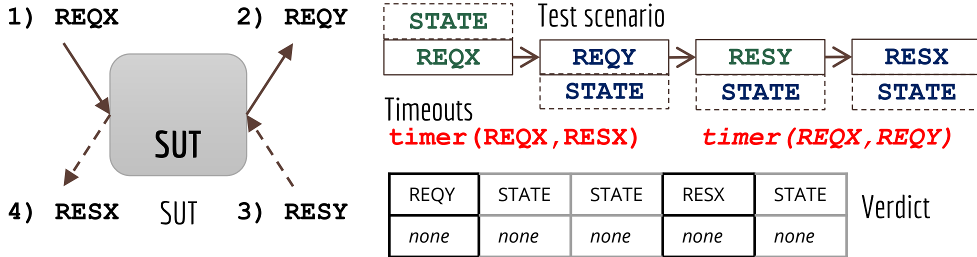
1. Optionally, the *client* sets the **actual SUT state** with the **canned SUT state** (contract-based testing)
2. The *client* sends to the SUT the **req stimulus request**
3. The *client* starts the **res timer** for the **res actual response** and waits - if **res timeout**, go to 6
4. The *client* receives the **res actual response** before the **res timeout**, stops the **res timer**, compares the **res actual response** with the **res oracle response**, and sets the **res response local verdict**
5. Optionally, the *client* retrieves the **actual SUT state**, compares it with the **oracle SUT state**, and sets the **SUT state local verdict** (contract-based testing)
6. The executor/arbiter re-initializes the SUT and returns the **compound test verdict**

 © 2017 – All rights reserved

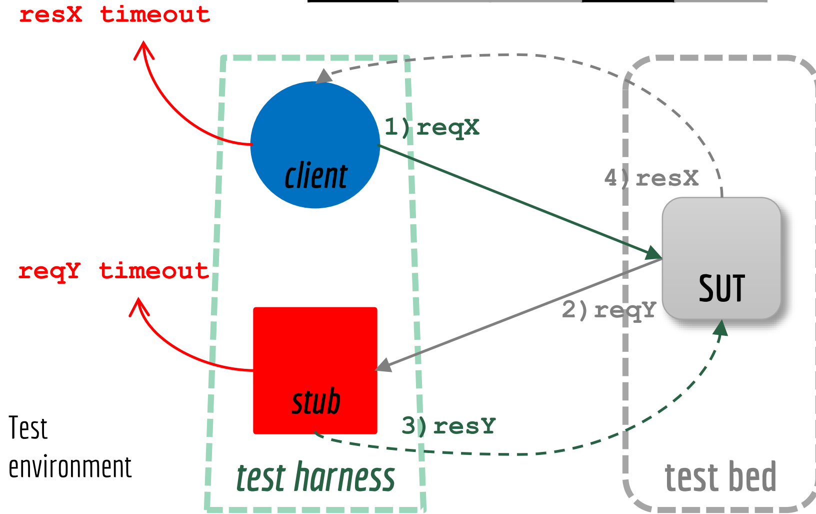
SIMPLY TESTIFY

Unit test of a non-terminal component

SIMPLY TESTIFY



REQY	STATE	STATE	RESX	STATE
none	*none*	*none*	*none*	*none*

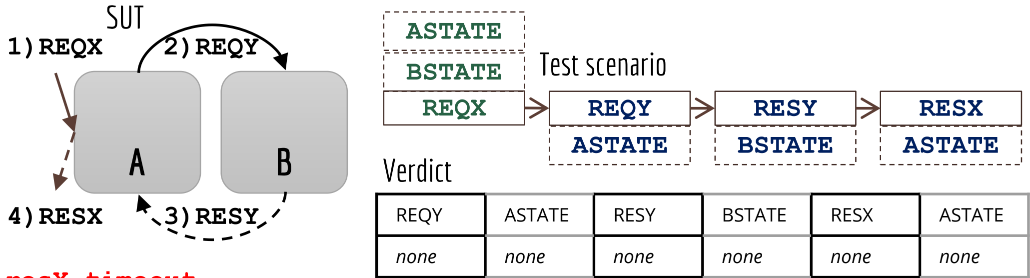


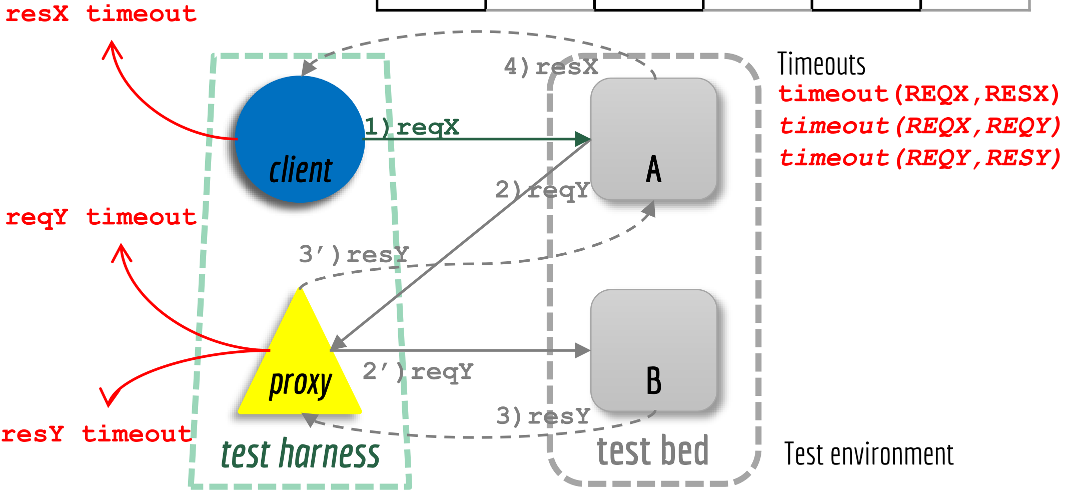
(*The SUT is initialized*)

1. Optionally, the *client* sets the **actual SUT state** with the **canned SUT state** (contract-based testing);
2. The *client* sends the **reqX stimulus request** to the SUT and notifies the *stub*
3. The *client* starts the **resX timer** and waits; if specified, the *stub* starts the **reqY timer** and waits; if one of the **timeouts** raises, go to 7
4. The *stub* receives the **reqY actual request** before the **reqY timeout**, stops the **reqY timer**, compares the **reqY actual request** with the **reqY oracle request**, and sets the **reqY message local verdict**; optionally, the *stub* retrieves the **actual SUT state**, compares it with the **oracle SUT state**, and sets the **SUT state local verdict** (contract-based testing); if one of the local verdicts is set to *fail*, go to 7
5. The *stub* sends the **resY stimulus response** to the SUT
6. The *client* receives the **resX actual response** before the **resX timeout**, stops the **resX timer**, compares the **resX actual response** with the **resX oracle response**, and sets the **resX message local verdict**;
7. Optionally, the *client* retrieves the **actual SUT state**, compares it with the **oracle SUT state**, and sets the **SUT state local verdict** (contract-based testing)
8. The executor/arbiter re-initializes the SUT and returns the **compound test verdict**

 © 2017 - All rights reserved

Integration test of a distributed Sut



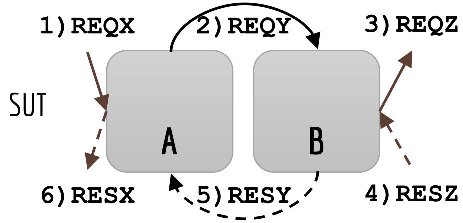


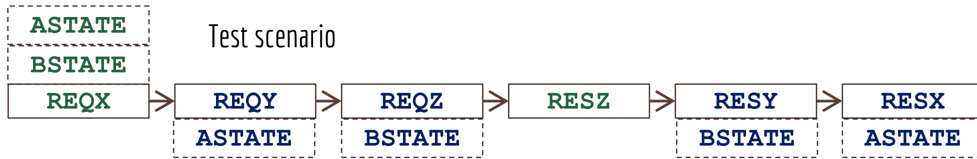
(*The SUT - A and B - is initialized*)

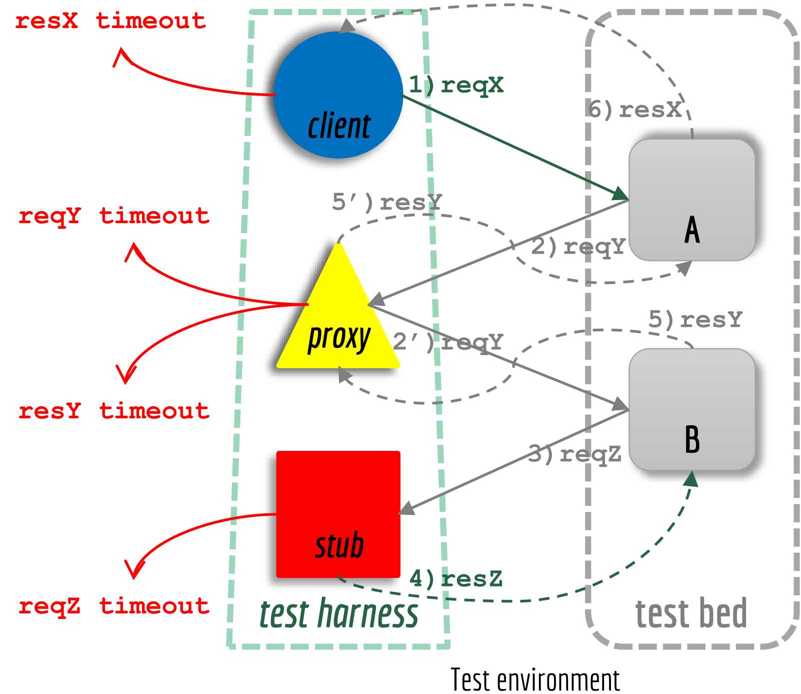
1. Optionally, the *client* sets the **actual A state** with the **canned A state**, the *proxy* sets the **actual B state** with the **canned B state** (contract-based testing)
2. The *client* sends the **reqX stimulus request** to A and notifies the *stub*
3. The *client* waits for the **resX actual response**, and the *proxy* waits for the **reqY actual request** - if one of the **timeouts** raises, go to 12
4. A sends **reqY actual request** to the *proxy*;
5. The *proxy* receives **reqY actual request**, compares it with the **reqY oracle request**, and sets the **reqY local test verdict** - optionally, it retrieves the **A actual state**, compares it with the **A oracle state**, and sets the **A state local verdict** (contract-based testing); if one of the local verdicts is set to fail, go to 12
6. The *proxy* sends **reqY actual request** to B and waits for the **resY actual response** - if **resY timeout** raises, go to 12
7. B sends **resY actual response** to the *proxy*
8. The *proxy* receives **resY actual response**, compares it with the **resY oracle response**, and sets the **resY local test verdict** - optionally, it retrieves the **B actual state**, compares it with the **B oracle state**, and sets the **B state local verdict** (contract-based testing); if one of the local verdicts is set to fail, go to 12
9. The proxy sends the **resY actual response** to A
10. A sends the **resX actual response** to the *client*
11. The *client* receives the **resX actual response**, compares it with the **resX oracle response** and sets the **resX local test verdict** - optionally, it retrieves the **A actual state**, compares it with the **A oracle state**, and sets the **A state local verdict** (contract-based testing)
12. The executor/arbiter re-initializes the SUT and returns the **compound test verdict**

© 2017 – All rights reserved

Integration test of a distributed Sut (II)







timeout(REQX,RESX)
timeout(REQX,REQY)
timeout(REQY,RESY)
timeout(REQY,REQZ)

Timeouts

REQY	ASTATE	REQZ	BSTATE	RESY	BSTATE	RESX	ASTATE
none	*none*	*none*	*none*	*none*	*none*	*none*	*none*

Verdict

© 2017 – All rights reserved