Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	

RIoT Quickstart guide

Contents

1 Introduction

This document is a quick and informal introduction to the RIoT Titan application.

RIoT is a load generator built on top of the TitanSim load generator framework. It is capable of simulating devices using some IoT protocols (CoAP[4], LwM2M[5], MQTT[6], HTTP). RIoT (and the TitanSim framework) was created to support non-functional tests, where load generation is required like performance, stability, scalability and so on.

2 Installation

2.1 Prerequisites

- Eclipse Titan
  The Titan TTCN-3 compiler is required to build the RIoT application
  https://projects.eclipse.org/projects/tools.titan

- Eclipse with TITAN TTCN-3 Plugins
  To navigate RIoT's source code it is recommended to install eclipse with the titan eclipse plugins

- Since the application will simulate several thousand devices it will want to use many simultaneously open network connections. In Linux, the ulimit command can be used to set resource limits on processes. Use ulimit -n to increase the maximum number of open file descriptors allowed for RIoT (100000 will be enough for the demos):

  ```
  ulimit -n 100000
  ```

2.2 Compiling RIoT

To clone RIoT's git repository:

```
git clone git://git.eclipse.org/gitroot/titan/titan.Applications.RIoT.git
```

To clone the submodule dependencies:

```
cd titan.Applications.RIoT
git submodule update –init
```

Generating the Makefile:

```
ttcn3_makefilegen -t RIOT_LPA108661.tpd
```

Compiling the source code:

```
cd bin
make dep
make
```

After the compilation was done successfully, a binary called “riot” can be found in the bin directory.

```
ttcn3@ttcn3-VirtualBox:~/riot/src/Applications/RIOT_LPA108661$ ./bin/riot -l
IOT_App_Functions.TC
```

3 Demos

In this section some load generator setups will be presented to demonstrate the RIoT application:

- CoAP performance test against the Californium stack (see 3.1)
- CoAP performance test against a simulated server (see 3.2)
- Simulated LwM2M devices against Leshan (see 3.3)
- Stability test against Leshan (see 3.4)

3.1 CoAP performance test against the Californium stack

In this setup the System Under Test (SUT) is a CoAP server realized with the Californium stack available from Eclipse[8]. The load generator is sending CoAP GET and POST requests to this server and waiting for a response. Execute the following steps to go through the demo.

To build Californium, please follow the instructions from here: https://github.com/eclipse/californium. After the build is successful, executable JARs of Californium's examples with all dependencies can be found in the demo-apps/run folder. We will use the `cf-plugtest-server` example as a System Under Test (SUT).

- Start the SUT
  - `cd <californium repo>/demo-apps/run`
  - `java -jar cf-plugtest-server-*.jar`
- Start RIoT
  - `cd <riot repo>`
  - `ttcn3_start ./bin/riot ./cfg/performance_californium/coap_basic_main.cfg`
  - To open RIoT's GUI you'll need to open a browser and go to http://127.0.0.1:4040
- Start Testing
  - On RIoT's GUI in the browser press the "Start Scenario"

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	

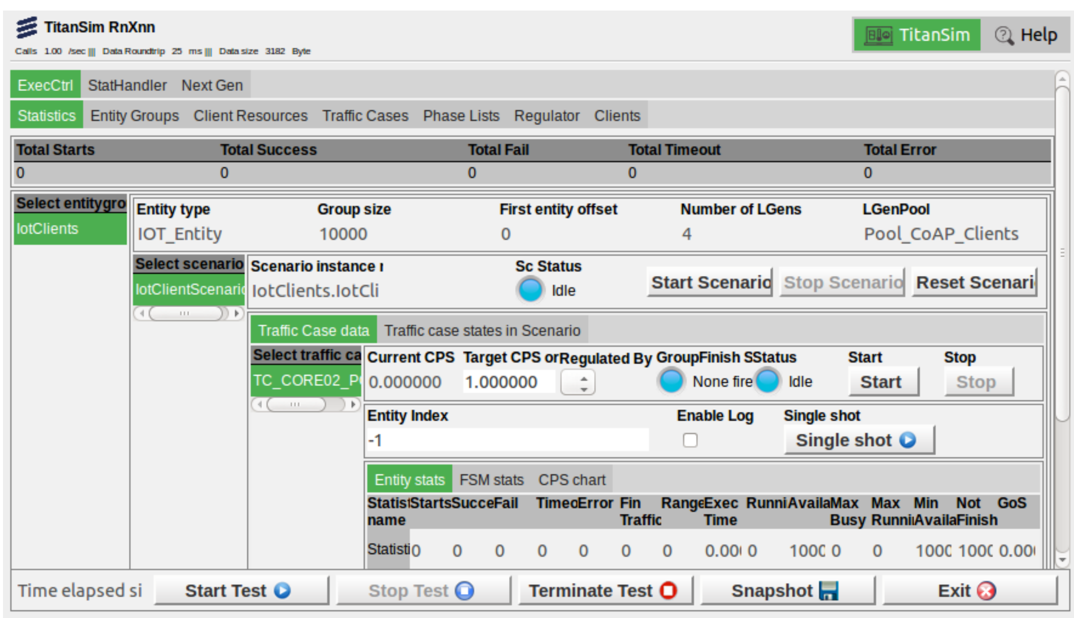
3.1.1 GUI

This section gives a short introduction to RIoT's GUI. After starting the application and opening http://127.0.0.1:4040, the Execution Control's Statistic page is shown in the browser. This page gives a good overview about the current test execution.

The entity group this demo is using is called "IoTClients". An entity group is a group of simulated devices. Here this group consists of 10.000 simulated IoT clients (Group size). These clients are distributed on 4 load generators (Number of LGens). This is important in case of performance testing, since each LGen is a physical Linux process. In order to utilize the cores available in the CPU, one need to create at least as many processes as CPU cores the executing machine has. This setup will use 4 cores.

On the entity groups one can start scenarios, where each scenario consists of traffic cases, where each traffic case is tied to a finite state machine that will implement the given traffic case. Here we are executing the "IoTClientScenario" which has only one traffic case: "TC_CORE_02_POST". The user can set a frequency to trigger the finite state machines tied to the traffic case by setting the "Target CPS" field. This field can be dynamically tuned even after the execution has been started.

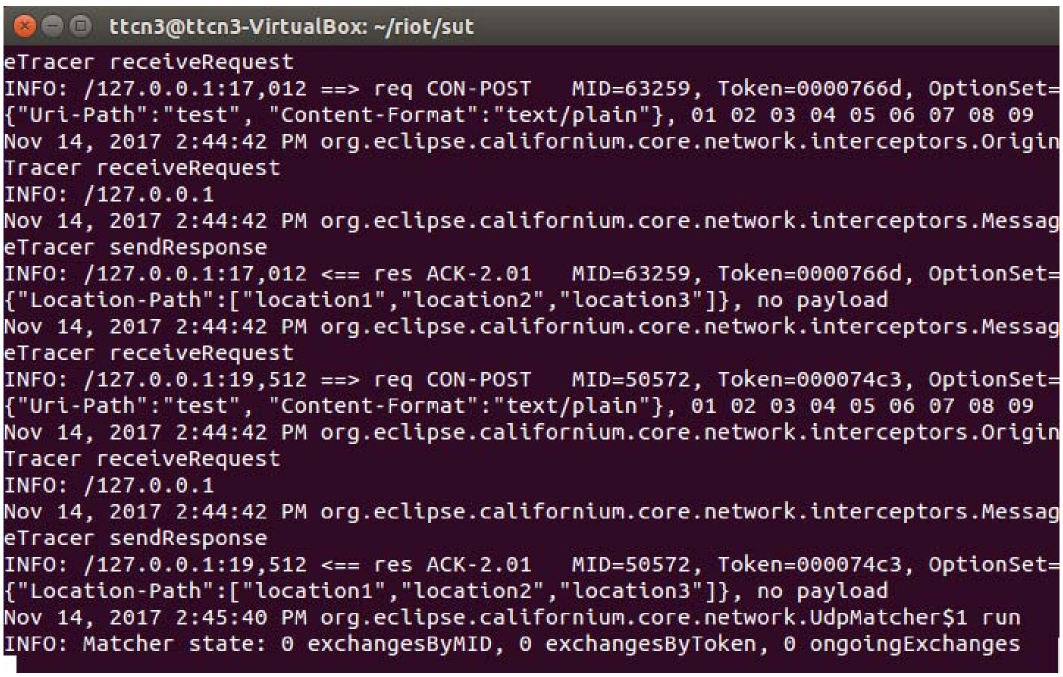
After the execution started, there are statistics presented. "Starts" is the number of triggers sent to the finite state machine instances. As the state machines are finishing their execution they will report back some verdict like pass, fail, timeout or error. There are counters assigned to these events and they are shown respectively on the GUI.



3.1.2 Californium

Californium is a CoAP protocol stack implementation. The sample application will log the messages received and sent to the console, where it was started.

In case you have Wireshark installed on your machine, you can trace the CoAP messages on the network interface (loopback interface). Look for UDP port 5683.



```
ttcn3@ttcn3-VirtualBox: ~/riot/sut
eTracer receiveRequest
INFO: /127.0.0.1:17,012 ==> req CON-POST   MID=63259, Token=0000766d, OptionSet=
{"Uri-Path":"test", "Content-Format":"text/plain"}, 01 02 03 04 05 06 07 08 09
Nov 14, 2017 2:44:42 PM org.eclipse.californium.core.network.interceptors.Origin
Tracer receiveRequest
INFO: /127.0.0.1
Nov 14, 2017 2:44:42 PM org.eclipse.californium.core.network.interceptors.Messag
eTracer sendResponse
INFO: /127.0.0.1:17,012 <== res ACK-2.01   MID=63259, Token=0000766d, OptionSet=
{"Location-Path":["location1","location2","location3"]}, no payload
Nov 14, 2017 2:44:42 PM org.eclipse.californium.core.network.interceptors.Messag
eTracer receiveRequest
INFO: /127.0.0.1:19,512 ==> req CON-POST   MID=50572, Token=000074c3, OptionSet=
{"Uri-Path":"test", "Content-Format":"text/plain"}, 01 02 03 04 05 06 07 08 09
Nov 14, 2017 2:44:42 PM org.eclipse.californium.core.network.interceptors.Origin
Tracer receiveRequest
INFO: /127.0.0.1
Nov 14, 2017 2:44:42 PM org.eclipse.californium.core.network.interceptors.Messag
eTracer sendResponse
INFO: /127.0.0.1:19,512 <== res ACK-2.01   MID=50572, Token=000074c3, OptionSet=
{"Location-Path":["location1","location2","location3"]}, no payload
Nov 14, 2017 2:45:40 PM org.eclipse.californium.core.network.UdpMatcher$1 run
INFO: Matcher state: 0 exchangesByMID, 0 exchangesByToken, 0 ongoingExchanges
```

3.1.3 Top window

The easiest way to check what kind of resources are used by the SUT and the load generator is to use top. RIoT has 4 “riot” processes that are used only for CoAP load generation. The java line in the screenshot below belongs to the Californium stack. There are other “riot” process that are either used for coordination or to handle specific functions like the REST API for the GUI. When you increase CPS only the load generator processes should require more resources.

Please keep in mind, that there are a number of factors that affect the performance of RIoT. Some examples:

- Turning on logging will result in great performance loss, as writing to disk is very slow. When logging is on, one must also be very careful to not run out of disk space.

- Optimizing the compilation has also a great effect on performance. Adding -O2 switch to the C++ compiler, will result in optimized binaries. It is also possible to turn on/off verbose logging statements in the code by using the preprocessor switch -DEPTF_DEBUG (The default generated Makefile in the bin directory is not optimized. You must add the -O2 switch to the CXXFLAGS= line and rebuild the executable)

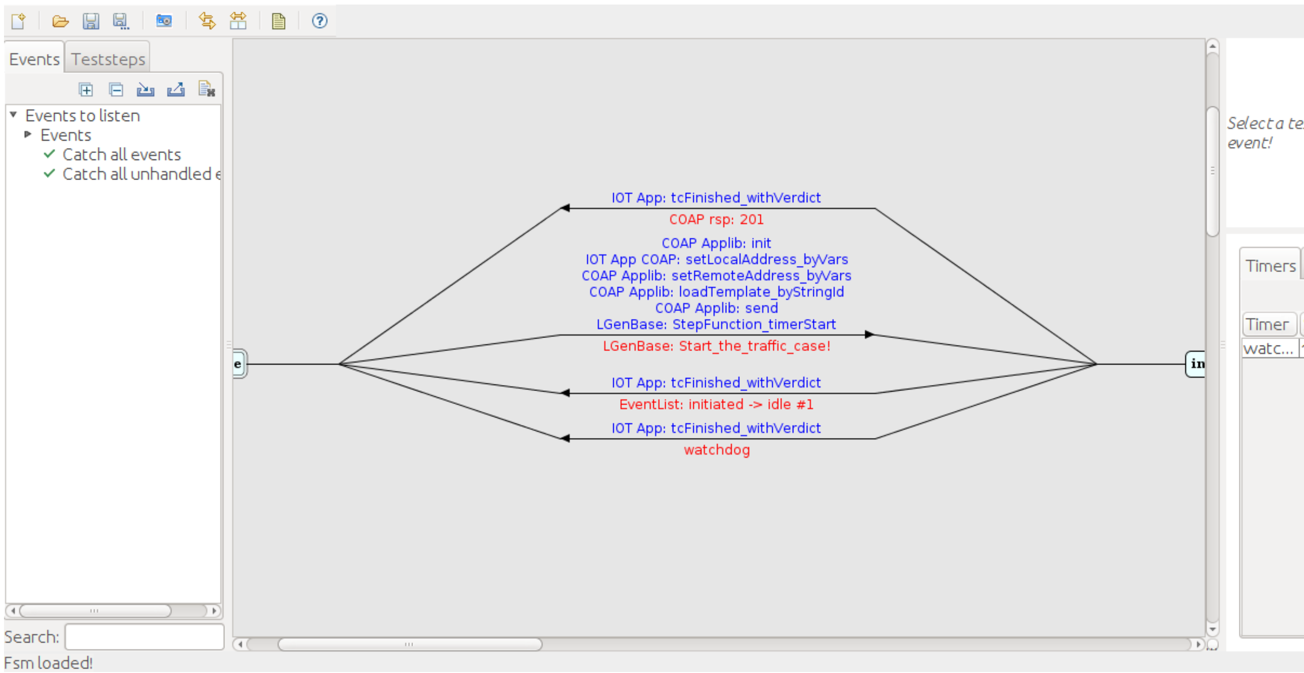
```
top - 14:44:26 up  1:34,  4 users,  load average: 1,39, 1,10, 0,97
Tasks: 181 total,   4 running, 177 sleeping,   0 stopped,   0 zombie
%Cpu(s): 70,1 us,  8,7 sy,  0,0 ni, 20,9 id,  0,0 wa,  0,0 hi,  0,3 si,  0,0 st
KiB Mem:   4135772 total,  1495348 used,  2640424 free,    63652 buffers
KiB Swap:  4191228 total,        0 used,  4191228 free.   665008 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2024 ttcn3     20   0  490216 223780  38684 R  74,5  5,4  46:26.17 compiz
 1024 root      20   0  186032  64004  11892 R  30,3  1,5  14:06.01 Xorg
 4745 ttcn3     20   0 1390796  70608   9940 S  20,0  1,7   0:08.56 java
 4866 ttcn3     20   0  533608 134136  39404 S  11,0  3,2   0:26.63 firefox
 2343 ttcn3     20   0  140524  23200  12568 R   9,0  0,6   1:00.57 gnome-term+
 4852 ttcn3     20   0   64068  15492   6796 S   3,3  0,4   0:00.78 riot
 4853 ttcn3     20   0   64100  15704   7004 S   2,7  0,4   0:00.76 riot
 4854 ttcn3     20   0   64064  15496   6796 S   2,3  0,4   0:00.75 riot
 4855 ttcn3     20   0   64068  15500   6796 S   2,3  0,4   0:00.74 riot
 1758 ttcn3     20   0   50856   8316   3476 S   1,7  0,2   0:25.76 ibus-daemon
 4850 ttcn3     20   0   57136   5360   3712 S   1,0  0,1   0:02.42 riot
 4851 ttcn3     20   0   56828   6520   5132 S   0,7  0,2   0:01.73 riot
   28 root      20   0       0      0      0 S   0,3  0,0   0:01.06 kworker/0:1
 1780 ttcn3     20   0  157624  15036  10848 S   0,3  0,4   0:01.38 unity-sett+
 4999 ttcn3     20   0    6916   1428   1036 R   0,3  0,0   0:00.02 top
    1 root      20   0    4456   2516   1424 S   0,0  0,1   0:02.87 init
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
```

3.1.4 State machines

As mentioned earlier, the state machine instances are implementing the behavior of the simulated devices. In this demo the state machines are specified in the configuration file. They can be found in <riot repo>/cfg/coap_basic/coap_basic_fsms.cfg and is called TC_CORE_02_POST_FSM. The description is text based, so it is possible to read and write it using a simple text editor. From the textual description a the following graphical representation can be drawn (see below).

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	



The FSM is very simple. It starts in the idle state. Then as the instance receives the “Start_the_traffic_case” event it will initialize its CoAP library and will load in a template (from coap_basic_templates.cfg”) to send it out. After executing these actions, it goes to the state called “initialized”. If it receives a CoAP 201 response in state “initialized” it finishes running and reports a pass verdict.

3.1.5 Configuration files

The configuration files for the load generator are in the <riot repo>/cfg/performance_californium directory. The configuration is described in four files:

- Coap_basic_main.cfg
- Coap_basic_params.cfg
- Coap_basic_fsms.cfg
- Coap_basic_tempates.cfg

3.1.5.1 Coap_basic_maing.cfg

This is the main configuration file. This is the one that should be passed as an argument when starting RIoT. It is including the rest of the configuration files using preprocessor include statements.

```
[INCLUDE]
```

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	

```
"coap_basic_params.cfg"
"coap_basic_fsms.cfg"
"coap_basic_templates.cfg"
```

The following structure will create 4 processes (load generators) to form a load generator pool "Pool_CoAP_Clients":

```
tsp_EPTF_ExecCtrl_LGenPool_Declarators :=
{
  {
    name := "Pool_CoAP_Clients",
    lgenPoolItems :=  { { hostname := "localhost", num := 4, createFunctionName :=
"RIoT.createLGen" } }
  }
}
```

This part below assigns the load generator pool to the entity group "IoTClients". You can assign several entity groups to a load generator pool. When a group is put on a load generator pool, it means that the elements (entities) of that pool will be distributed on the load generator pool. Using this construct, it is possible to distribute the simulated entities of a group on more than one cores of the host and this enables to generate larger loads (higher calls per seconds)

```
tsp_EPTF_ExecCtrl_EntityGroup2LGenPool_List :=
{
  {
    eGrpName := "IotClients",
    lgenPoolName := "Pool_CoAP_Clients"
  }
}
```

The next part describes what the IoTClients entity group is. This group contains 10.000 instances of the entity type "IOT_Entity". The entity type IOT_Entity is defined using TTCN-3 code (in IOT_LGen_Functions.ttcn) and it describes what kind of application libraries (behaviors) the given entity can use.

```
tsp_LGenBase_EntityGrpDeclarators := {
  { name := "IotClients", eType := "IOT_Entity", eCount := 10000 }
}
```

The next part is defining a scenario "IoTClientScenario", which has only one traffic case using a state machine called "TC_CORE02_POST_FSM" and assigns this scenario to the "IotClients" entity group.

```
tsp_EPTF_ExecCtrl_Scenario2EntityGroupList := {
  { scenarioName := "IotClientScenario", eGrpName := "IotClients", name := omit}
}

tsp_LGenBase_TcMgmt_tcTypeDeclarators2 := {
  {
    name := "TC_CORE02_POST",
    fsmName := "TC_CORE02_POST_FSM",
    entityType := "IOT_Entity",
    customEntitySucc := ""
  }
}

tsp_LGenBase_TcMgmt_ScenarioDeclarators3 :=
{
```

```
    {
      name := "IotClientScenario",
      tcList := {
        {
          tcName := "TC_CORE02_POST",
          tcParamsList := {
            {startDelay := 1.0},
            {target := { cpsToReach := 1.0 }},
            {scheduler := {preDefinedName := cs}},
            {enableEntitiesAtStart := true},
            {enabledAtStart := true}
          }
        }
      },
      scParamsList := {
        {enabled := true}
      }
    }
  }
```

This a very simple setup. But this hierarchic structure (of scenarios, traffic cases and entity groups) allows the users of TitanSim to create sophisticated traffic mixes. For more details, please look at the LGenBase documentation.

3.1.5.2 Coap_basic_params.cfg

Here mostly those switches are provided that turn on and off certain log levels for different components.

In case you plan to generate high load and/or load for a longer time, it is recommended to disable logging TTCN_USER and TTCN_ACTION to the log files, otherwise the system will run out of HDD space quickly.

```
[LOGGING]
FileMask :=  TTCN_ERROR | TTCN_TESTCASE | TTCN_STATISTICS | TTCN_WARNING | TTCN_ACTION #|
LOG_ALL #| DEBUG
ConsoleMask := TTCN_ERROR | TTCN_TESTCASE | TTCN_STATISTICS | TTCN_WARNING  #| TTCN_ACTION
```

3.1.5.3 Coap_basic_fsms.cfg

This file contains the FSM descriptions.

3.1.5.4 Coap_basic_templates.cfg

Message structures are described here that are used by the state machines described in coap_basic_fsms.cfg. An example:

```
{
  id := "t_TC_CORE_01_GET",
  msg :=
  {
    header :=
    {
      version := 1,
      msg_type := CONFIRMABLE,
```

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	

```
          code := { class:= 0, detail := 1 },
          message_id := 0
        },
        token := ''O,
        options :=
        {
          {
            uri_path := "test"
          }
        },
        payload := omit
      }
    },
```

3.2 CoAP performance test against a simulated server

In this setup both the CoAP server and the CoAP clients are simulated by RIoT. This is intended to be an example of how to use separate groups and different behaviors in the same configuration.

- Start RIoT
  - `cd <riot repo>`
  - `ttcn3_start ./bin/riot ./cfg/coap_basic/coap_basic_main.cfg`
  - To open the GUI you'll need to open a browser and go to http://127.0.0.1:4040
- Start Testing
  - On RIoT's GUI
    - Select "IoTServer" entity group by clicking on it.
    - Push the "Start Scenario", to start the CoAP server
    - Select "IoTClients" entity group by clicking on it.
    - Push the "Start Scenario", to start the CoAP clients
    - You can change the desired cps during running.
- Stop testing
  - On RIoT's GUI
    - Select IotClients entity group
    - Push Stop scenario button to stop the clients
    - Select IoTServer entity group

        - Push Stop scenario button to stop the server
        - Press ‘Exit’ to exit from RIoT

3.3 Simulated LwM2M devices against Leshan

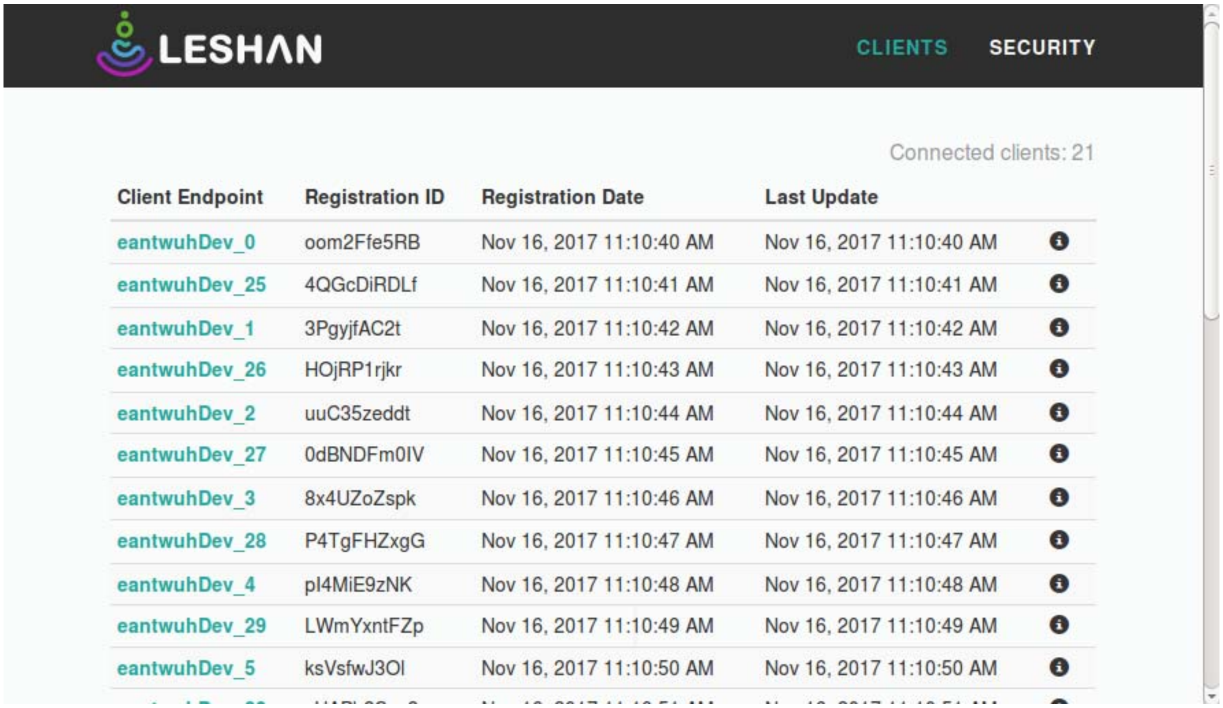
In this setup the System Under Test (SUT) is a Lightweight Machine2Machine (LwM2M) server realized with the Leshan library available from Eclipse[9]. The load generator is simulating LwM2M devices that are registering in to the server, publishing their smart objects and finally deregistering.

Leshan sources and precompiled JAR packages can be found from: https://github.com/eclipse/leshan. For this demo the leshan-server-demo.jar package will be needed.

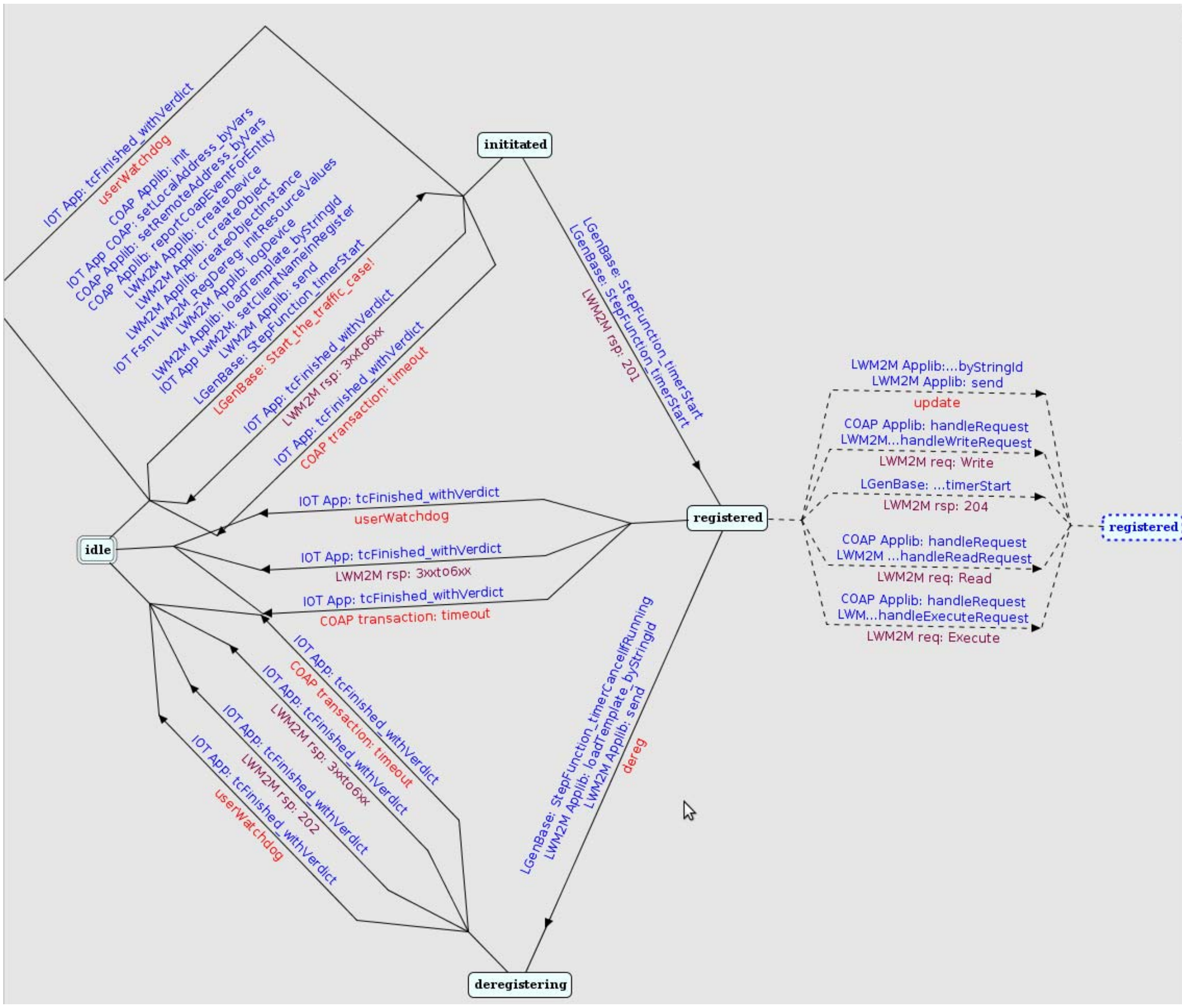
- Start Leshan
    - java -jar leshan-server-demo.jar
    - Open the Leshan GUI using a browser http://127.0.0.1:8080
- Start RIoT
    - `cd <riot repo>`
    - `ttcn3_start ./bin/riot ./cfg/leshan_basic/leshan_basic_main.cfg`
    - To open the GUI you’ll need to open a browser and go to http://127.0.0.1:4040
- Start Testing
    - On RIoT’s GUI
        - Select “IoTClients” entity group by clicking on it.
        - Push the “Start Scenario”, to start the LwM2M clients

After starting the device simulation, the clients will register in to the Leshan LwM2M server. On Leshan’s GUI you should see them listed:

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	**Checked**	**Date**	**Rev**	**Reference**
		2018-03-02	PA1	



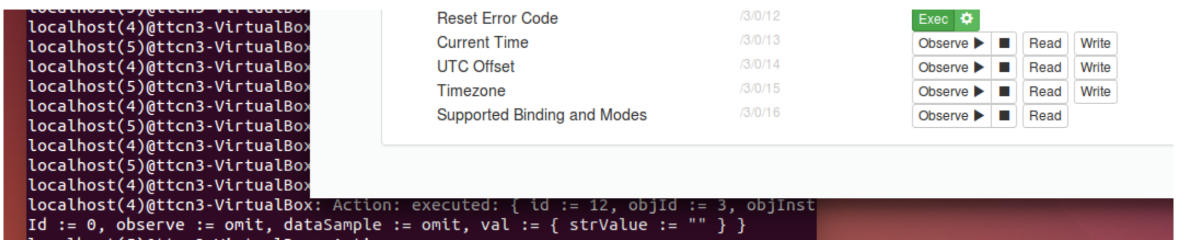
The clients' behavior is in the leshan_basic_fsms.cfg, the FSM is called "LWM2M_RegDereg_FSM". The simulated devices are registering in. Then keep their registration alive by reregistering for a while and finally they deregister.



The LwM2M library is capable to handle some incoming requests as well, but not all LwM2M procedures are supported currently. Some examples:

- Reading a smart object value
    - On the Leshan GUI select a device
    - Locate the smart object Device/instance 0/Manufacturer and push the Read button
    - This will send a CoAP read request to the simulated device which will answer with a response

- Executing a smart object value
  - o On the Leshan GUI select a device
  - o Locate the smart object Device/instance 0/Reset error code and push the Execute button
  - o This will send an execute request to the simulated device which will receive it and answer it. You shall notice in RIoT's terminal a message, that indicates the request was received.



- Stopping the test
  - o On RIoT's GUI select "IoTClient" entity group and push stop scenario.
  - o After a few seconds all the simulated devices will deregister (you can check this on Leshan's GUI)
  - o Push Exit on RIoT's GUI

3.4 Stability test against Leshan

This case is a variation for the previous one (3.3). The SUT is Leshan again. RIoT is simulating a little bit more devices than before (1000) and uses a little bit larger calls per second to start them up (50cps). This is to demonstrate that the system can simulate large numbers of entities. If you check top you shall see that the simulation still does not consume too much resources. (With other applications built with TitanSim we were able to simulate ~1M SIP signaling phones with media generation on PC hardware)

- Start Leshan
  - o java -jar leshan-server-demo.jar

  - It is not recommended to open Leshan’s GUI (actually you should close its tab in firefox before starting up RIoT), because it may have problems handling these many devices!

- Start RIoT
  - `cd <riot repo>`
  - `ttcn3_start ./bin/riot ./cfg/stability_leshan/leshan_basic_main.cfg`
  - To open the GUI, you’ll need to open a browser and go to http://127.0.0.1:4040

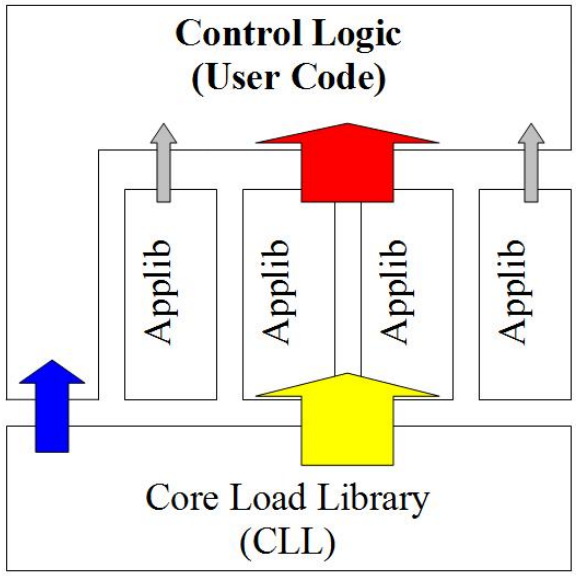
- Start Testing
  - On RIoT’s GUI
    - Select “IoTClients” entity group by clicking on it.
  - Push the “Start Scenario”, to start the LwM2M clients

When the execution is started, all 1000 devices will register in with 50cps. But this time they won’t deregister automatically, instead they keep alive their registration by reregistering. Which means about 50 reregisters per second during test execution. To idea with this stability test is to keep them running for a long time to see, if the SUT is stable enough to handle this load.

- Stopping the test
  - On RIoT’s GUI select “IoTClient” entity group and push stop scenario.
  - After a few seconds all the simulated devices will deregister
    (The deregistration is not distributed in time. The devices will get the stop event and they try to immediately deregister at once, thus creating a peak load)
  - Push Exit on RIoT’s GUI

4 Source code

The source code is divided into several components, where each component is mapped to a directory. To help understanding the arrangement of the components in the software one must know how a TitanSim application is constructed. The TitanSim framework is a 3-layered software framework aimed at developing TTCN-3 load test applications.



The three layers are defined as follows:

1 Core Load Library (CLL)
`<riot repo>/src/Libraries/EPTF_Core_Library_CNL113512`
This library realizes a common base foundation for the whole framework and provides project, SUT and protocol independent functionality

2 The various Application Libraries (AppLib)
`<riot repo>/src/Libraries/EPTF_Applib_*`
They are usually protocol, or application-area dependent, but can be *reused* across many TitanSim applications

3 The Application level code (often called as Control Logic) that “glues” together the various framework components:
`<riot repo>/src/Libraries/IoT_LoadTest_Framework`

3.1 Configuration logic (what can and must be configured and what is set implicitly, what is configured statically and what can be set interactively, etc.)

3.2 Statistics generation and collection logic (what data is generated, how the data is reported and which data is recorded in logs and which is displayed during execution, etc.)

3.3 Deployment logic (which software component is deployed to which PTC, whether distributed execution of a given Entity Group is supported, or not, etc.

Prepared (Subject resp)		No.		
Antal Wu-Hen-Chang				
Approved (Document resp)	Checked	Date	Rev	Reference
		2018-03-02	PA1	

5 References

[1] Oracle VirtualBox
https://www.virtualbox.org/

[2] Ubuntu Linux 14.04.1 Desktop i386
http://old-releases.ubuntu.com/releases/14.04.1/

[3] Titan TTCN-3 Test Executor
https://projects.eclipse.org/projects/tools.titan

[4] CoAP protocol
http://coap.technology/

[5] LwM2M protocol
http://openmobilealliance.org/iot/lightweight-m2m-lwm2m

[6] MQTT protocol
http://mqtt.org/

[7] Eclipse
http://www.eclipse.org

[8] Eclipse Californium
https://www.eclipse.org/californium/

[9] Eclipse Leshan
https://www.eclipse.org/leshan/