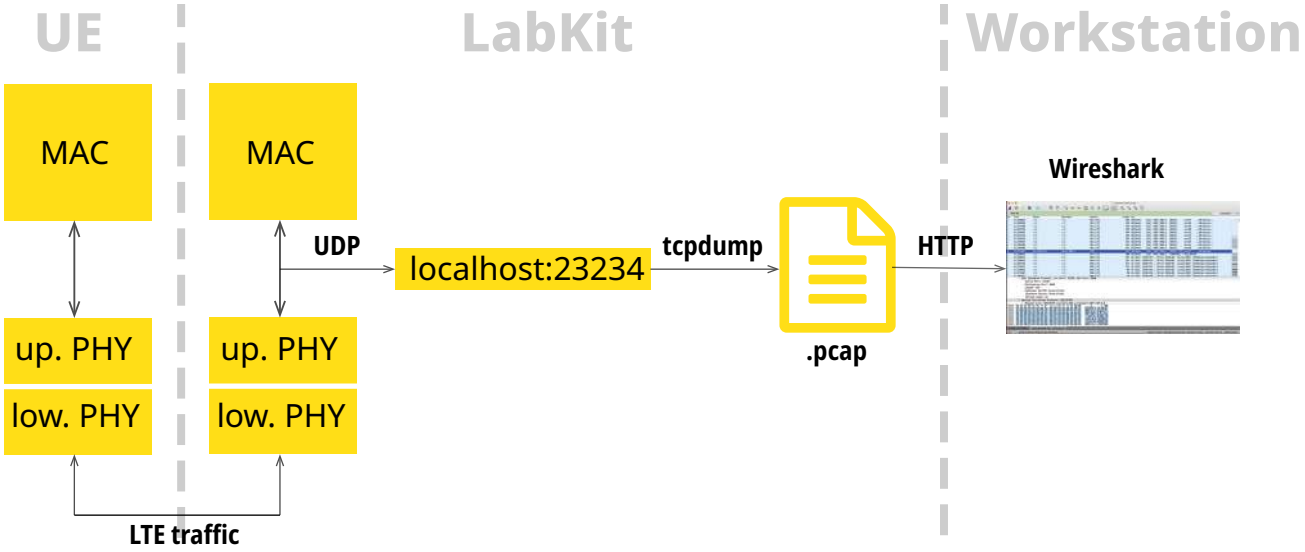
GSM/LTE LabKit

APPLICATION NOTES

EASY CONFIGURABILITY ■ WIRESHARK-READY ■ LINUX-BASED ■ GREAT MONITORING CAPABILITIES FOR RADIO TRAFFIC

APPLICATION NOTE APPN001 2018-02-01

Wireshark LTE Analysis on GSM/LTE LabKits



*by **David A. Burgess***

One of the most useful features of LTE LabKits and SatSites is their ability to monitor traffic, both between the eNodeB and the UE and the EPC and the eNodeB. This is done with a mix of YateENB and Unix commands and results in a capture compatible with analysis tools such as Wireshark. This application note describes the procedures and results of monitoring, from the LabKit/SatSite owner's point of view. Throughout this guide, we'll use LabKit as a name for the Yate/YateBTS eNB, but the tutorial fully applies to our SatSite line of products.

I. PRE-REQUISITES

In order to be able to read the capture file you have to have the cyphering turned off, which means that the UE, LabKit and MME have to use the **EEA0** EPS encryption algorythm. In case you are using YateBTS Hosted Core, YateBTS Minicore or YateUCN, this is provided to you by default. In case you are using another core, please contact your provider to switch to EEEA0.

II. CONNECTING AND PERFORMING THE CAPTURE

1. Connect to the LabKit by **ssh**:

```
ssh yatebts@YOUR_LABKIT_IP -p 54321
```

The password is **the serial number** printed on the front plate on your LabKit.

2. Prepare a work directory on the LabKit WWW server. This step is necessary in order to subsequently get the capture file on your workstation. (You only need to do this step once.)

2.1. Go to the Web root:

```
cd /var/www/html
```

2.2 Switch to **root** (same password as the yatebts user) and create a directory with a meaningful name, such as **pcap** or **wireshark**, in your web root. Go to the directory, to create the capture file there:

```
su
mkdir YOUR_DIRECTORY
cd YOUR_DIRECTORY
```

3. Telnet to be able to access the YateENB rmanager commands, on port **5037**:

```
telnet localhost 5037
```

4. Type in the following **rmanager** command:

```
enb capture start mac 23234
```

This will route the radio traffic to UDP port **23234**. You should get an `OK` answer from the **rmanager.**

5. Start and stop the actual capture:

Exit Telnet (`quit` or `CONTROL + C`). Make sure you are root to be able to initiate the capture: `su` on LabKits, with the same password as the yatebts user.

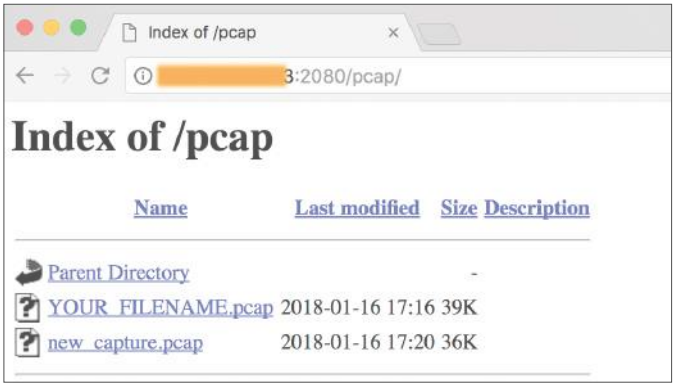
```
tcpdump -i any not tcp -w YOUR_FILENAME.
pcap
```

Start using the UE. When you're done, abort the capture with `CONTROL + C`. The capture should be in the root of your LabKit Web server.

6. Transfer the file to your workstation:

In order to be able to perform the analysis you have to transfer the capture file to your workstation. To do so, type:

`YOUR_LABKIT_IP:2080/YOUR_DIRECTORY` in your **WWW browser** location bar.



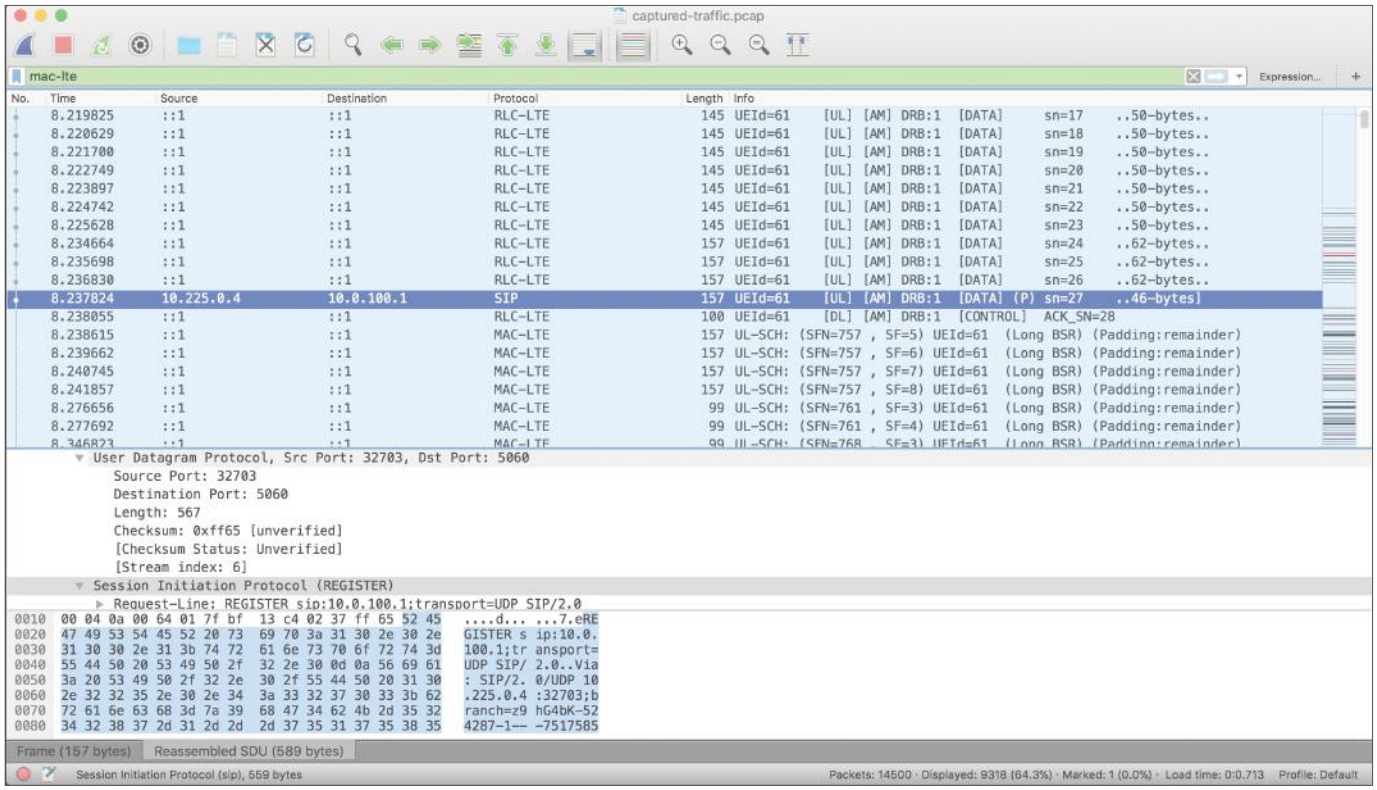
Now you should be able to click on your file and download it on your workstation.

III. WIRESHARK ANALYSIS

The capture includes information both about the traffic **between the LabKit and the UE**, and t**he LabKit and the MME**. You simply **Open** it from the **File** menu in Wireshark. To see the **MAC-LTE traffic** you have to enable all the MAC-LTE protocols from the **Analyze** menu. Also, set in **Preferences/Protocols/Mac-LTE:**

- Source of LCID -> drb chanel settings: **From your configuration protocol**
- Which Layer info to show in Info column: **RLC info**.

An example should look like this:



NOTES

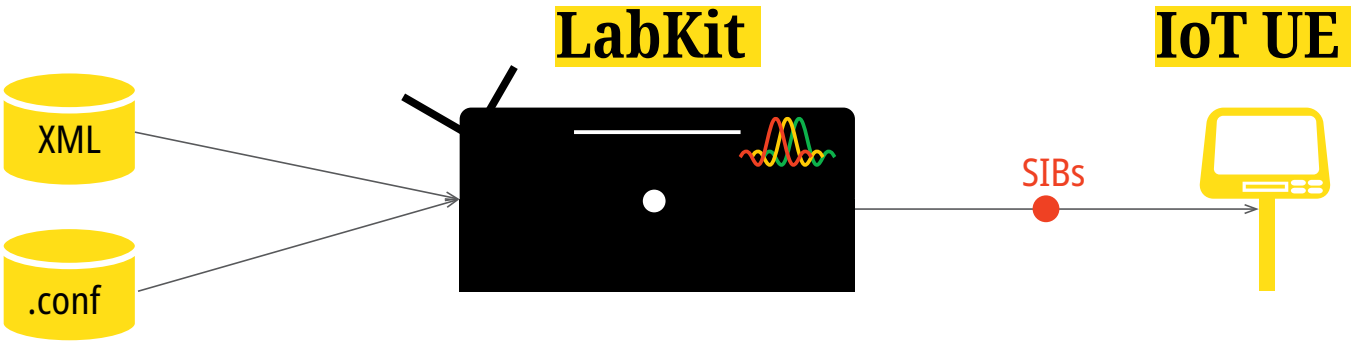
A. *The LabKit does have a GUI and Wireshark pre-installed so you could either perform the analysis locally or use the LabKit Wireshark by means of ssh X-forwarding. However,* ***we advise against using the LabKit as a workstation.*** *Also, you can use sftp or scp to get the file locally, but you have to change the permissions of the capture file.*

B. *In some Wireshark versions, radio trafic is inadvertently classified as "SKYPE". To see the actual protocol, disable Skype in* ***Analyze/ Enabled protocols.***

C. *This Application Note was built starting from public documentation written by* ***SS7Ware****.*

APPLICATION NOTE APPN002 2018-02-08

GSM/LTE LabKit easy configuration for IoT tests



*by **David A. Burgess***

The GSM/LTE LabKit is very easily configurable for a wide range of purposes, including the most uncommon setups necessary for debugging and running virtually any IoT application. Advertising of all kind of configurations from the LabKit to UEs is achieved with plain XML files which are transformed into binary SIBs (System Information Blocks).

XML files may include any information elements defined in the **Release 12** 3GPP specification **(3GPP 36.331).** Granular control of the LTE LabKit is useful for a variety of purposes, such as debugging a prototype IoT UE. By using Wireshark, Yate/YateBTS customers are able to get the transport blocks above the physical layer of communication between the UE and the eNB.
The XML files can easily be edited via SSH and/or locally, due to the stand-alone computer architecture of the LTE LabKit.

OVERVIEW

System Information Blocks (SIBs) are broadcast messages that advertise the configuration of the eNodeB.
The **YateENB** module generates the binary content of its SIBs **from XML files.** The module parses these files whenever it is started or restarted. Any information elements defined in the Release 12 RRC specification (3GPP 36.331) can be encoded into these files, regardless of whether the associated features are actually supported in the YateeNB module.
The XER-UPER encoder for these files is generated directly from the **ASN.1 source code** in the specifications. The syntax of the XML files is **standard ASN.1 XER,** except that the "-" character is substituted with "_" in element names and enumerated values.
For SIB Type 1, there are two possible XML files, one for FDD and one for TDD:

- **enb-sib1.fdd.xml**
- **enb-sib1-tdd.xml**

For all other SIB types there is a single possible XML file named **enb-SIB<N>.xml,** where <N> is the SIB type.
In a LabKit or SatSite system, these files are located in:
/usr/share/yate/scripts.

***Note:** Advertising features that the eNodeB does not support may cause UE connections to fail. Understand the*

implications of any changes to the SIB content before making those changes. Save copies of the default, original SIB XML files before changing them.

VARIABLE SUBSTITUTION

The XML encoder can set the values of SIB information elements from the general section of the **YateeNB.conf** or **YateeNB-custom.conf** configuration file. These values are substituted into the XML as strings before the XML-to-PER encoding step.
For example, here is the standard FDD SIB Type 1 XML file distributed with YateeNB:

```
<systemInformationBlockType1>
  <cellAccessRelatedInfo>
    <plmn_IdentityList>
      <PLMN_IdentityInfo>
        <plmn_Identity>
          <mcc>${MCC}</mcc>
          <mnc>${MNC}</mnc>
        </plmn_Identity>
        <cellReservedForOperatorUse>notReserved</cellReservedForOperatorUse>
      </PLMN_IdentityInfo>
    </plmn_IdentityList>
    <trackingAreaCode>${TAC}</trackingAreaCode>
    <cellIdentity>${CellIdentity}</cellIdentity>
    <cellBarred>notBarred</cellBarred>
    <intraFreqReselection>allowed</intraFreqReselection>
    <csg_Indication>false</csg_Indication>
  </cellAccessRelatedInfo>
  <cellSelectionInfo>
    <q_RxLevMin>${RxLevMin}</q_RxLevMin>
  </cellSelectionInfo>
  <freqBandIndicator>${Band}</freqBandIndicator>
  <schedulingInfoList>
    <SchedulingInfo>
      <si_Periodicity>rf${SiPeriodicity}</si_Periodicity>
      <sib_MappingInfo>
          <SIB_Type>sibType3</SIB_Type>
          <SIB_Type>sibType4</SIB_Type>
      </sib_MappingInfo>
    </SchedulingInfo>
  </schedulingInfoList>
  <si_WindowLength>ms${SiWindowLength}</si_WindowLength>
  <systemInfoValueTag>0</systemInfoValueTag>
</systemInformationBlockType1>
```

In this example, the following elements take their values from the YateeNB module configuration:

ELEMENT NAME	CONFIG. PARAMETER
mcc	MCC
mnc	MNC
trackingAreaCode	TAC
cellIdentity	CellIdentity
q_RxLevMin	RxLevMin
freqBandIndicator	Band
si_Periodicity	“rf” + SiPeriodicity
si_WindowLength	“ms” + SiWindowLength

These particular configuration parameters are either defined in YateeNB.conf by the Mobile Management Interface (MMI) or Local Management Interface (LMI), or are related to default values in the YateeNB module itself during initialization.
You can also add new parameters to **YateeNB-custom.conf** and then use them in the SIB XML files in this way. The only restrictions are that he parameter name cannot conflict with any existing YateeNB parameter.
The new element or element value does not conflict with the actual implementation of the eNodeB.
So, for example, to make cell-barring configurable, change the line:

```
<cellBarred>notBarred</cellBarred>
```

to something like:

```
<cellBarred>${cellBarredParam}</cellBarred>
```

and then add to **YateeNB-custom.conf:**

```
; Cell Barred configuration in SIB1.
; Valid values are “notBarred” and “barred”.
callBarredParam notBarred
```

Note 1: *If you try to add new parameters to yatenb.conf instead of YateeNB-custom.conf, the LMI or MMI will erase them the next time it updates the configuration.*

Note 2: *Parameters that are hard-coded in the SIBs provide reasonable default values in most cases. Changing them may degrade performance.*

ERROR CHECKING

The YateeNB module logs XML/SIB encoding errors **at the CRIT level.** If you edit the XML files, be sure to check the logs for encoding error warnings the first time you run the eNodeB.

ADDING NEW SIBS

The YateeNB module schedules all SIBs Type 2 and higher in a single SI Message, with the same periodicity. To add a new SIB type `<N>` to the eNodeB:

- Create the new XML file **enb-sib<N>.xml** in **/usr/share/yate/scripts.**
- Define a parameter `SibList` in **YateeNB-custom.conf,** which lists all of the SIBs (above SIB Type 2) that are to be used.
  - The form for the SibList parameter is **sibType3 sibType4,** etc.
  - The presence of SIB Types 1 and 2 is implied, so those do not appear in the list.

EXAMPLE 1

For example, to add a simple SIB Type 4 with one neighbor, define a new file **enb-sib4.xml,** with this content:

```
<sib4>
    <intraFreqNeighCellList>
        <IntraFreqNeighCellInfo>
        <!-- These values will be pulled from the configuration when the message is gen-
erated -->
            <physCellId>${neighborPhyCellID}</physCellId>
            <q_OffsetCell>dB${neighborOffset}</q_OffsetCell>
        </IntraFreqNeighCellInfo>
    </intraFreqNeighCellList>
</sib4>
```

and add these line to YateeNB-custom.conf:

```
; List of supported SIBs above Type 2
```

```
  SibList = sibType3 sibType4
  ; example neighbor in SIB4
  ; PHY cell ID
  neighborPhyCellID = 5
  ; neighbor power offset in dB
  neighborOffset = 0
```

Then edit **enb-sib1.tdd.xml** and **enb-sib1.fdd.xml** to update the SIB Mapping Information element:

```
      <sib_MappingInfo>
      <!-- The SIB list must be updated here and also in the .conf file. -->
          <SIB_Type>sibType3</SIB_Type>
          <SIB_Type>sibType4</SIB_Type>
      </sib_MappingInfo>
```

EXAMPLE 2

In this example, we add two new XML files, **enb-sib10.xml** and **enb-sib11.xml,** intended for use in an ETWS (Earthquake and Tsunami Warning System).

For **enb-sib10.xml:**

```
<sib10>
        <!-- These bit strings and byte strings are all given in hexadecimal. -->
        <!-- earthquake and tsunami -->
        <messageIdentifier>1101</messageIdentifier>
        <serialNumber>4001</serialNumber>
        <!-- earthquake, alert user, pop-up window -->
        <warningType>0180</warningType>
</sib10>
```

For **enb-sib11.xml:**

```
<sib11>
        <!-- These bit strings and byte strings are all given in hexadecimal. -->
        <!-- earthquake and tsunami -->
        <messageIdentifier>1101</messageIdentifier>
        <serialNumber>4001</serialNumber>
        <warningMessageSegmentType>lastSegment</warningMessageSegmentType>
        <warningMessageSegmentNumber>0</warningMessageSegmentNumber>
        <!-- 42-character message: “This is a test of ETWS. This is only a test. :)” -->
        <!-- The fields are defined in 3GPP 23.041: 1 byte page number, 82 bytes message
content (padded), 2 bytes mesage length (in bytes) -->
        <warningMessageSegment>0154747A0E4ACF416110BD3CA-
783DE6650917A9DBA4054747A0E4ACF416F373B0F0A83E8E539DD05D2A514D46A3D168341A-
8D46A3D168341A8D46A3D168341A8D46A3D168341A8D46A3D168341A8D46A3D16002A</warningMessageSeg-
ment>
        <!-- GSM 7-bit coding scheme, English language-->
        <dataCodingScheme>01</dataCodingScheme>
</sib11>
```

Now, you have to edit **enb-sib1.tdd.xml** and **enb-sib1.fdd.xml** to update the SIB Mapping Information element:

```
      <sib_MappingInfo>
          <!-- SIBs are listed here and in yateenb.conf. -->
          <SIB_Type>sibType3</SIB_Type>
          <SIB_Type>sibType10</SIB_Type>
          <SIB_Type>sibType11</SIB_Type>
      </sib_MappingInfo>
```

And also set the SIB list in yateenb.conf:

```
; List of supported SIBs above Type 2
SibList = sibType3 sibType4
```

MESSAGE SIZE LIMITATION

All SIBs Type 2 and higher **must fit together into a single SI Message,** in a single subframe, using QPSK modulation. The YateENB module will lower the coding rate for SIBs as needed, to try to fit all of these SIBs into the subframe. If the SIBs do not fit into the subframe even at the lowest allowed rate, the eNodeB will log a warning at the CRIT level.

Approximate maximum message sizes are:

LTE BANDWIDTH (MHZ)	APPROX. MAX. SI MESSAGE (KBITS)
1.4	1.4
3	3.5
5	5.9
10	12
15	18
20	24

The YateeNB module may also change the DCI type for SIBs to increase the maximum message size. It will log a message at the CONF level if it does this.

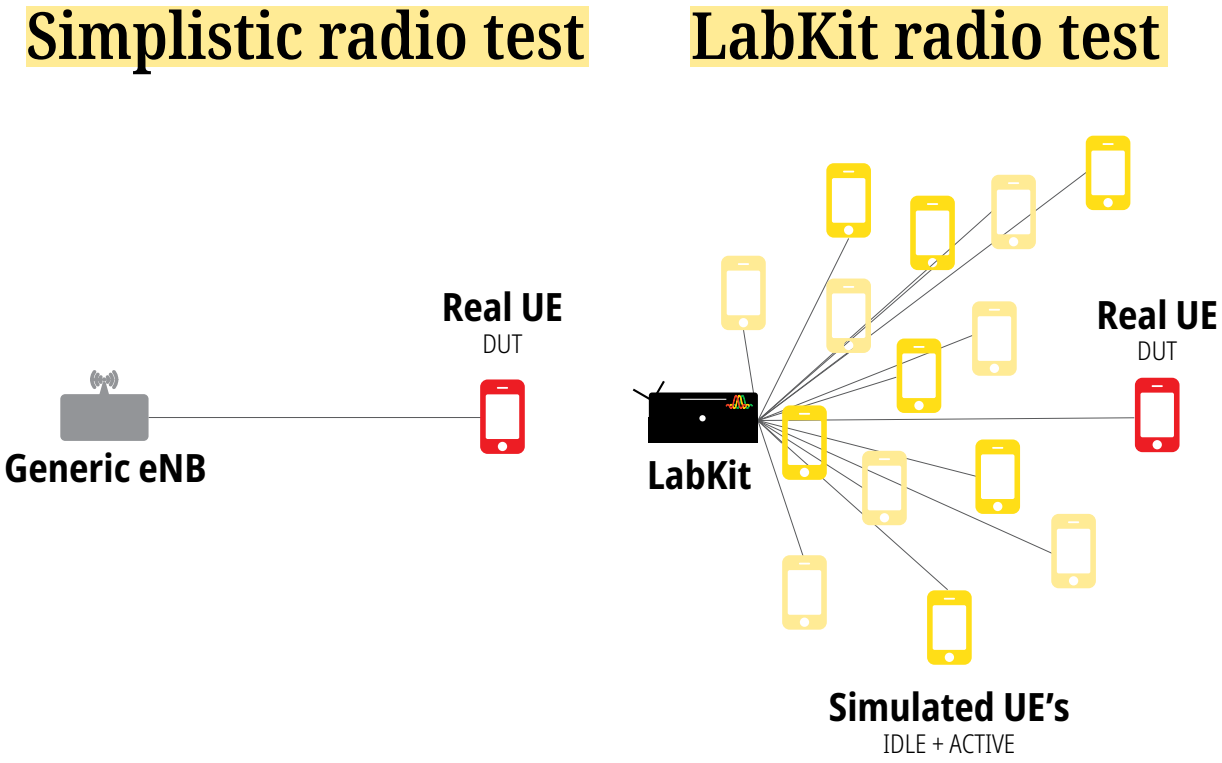
APPLICATION NOTE APPN003 2018-02-15

Simulating a complex LTE Network with the GSM/LTE LabKit

by ***David A. Burgess***

One of the most sought after features of eNodeB laboratory equipment is the ability to replicate various LTE real-life situations, in order to see how a test UE is affected by different levels of activity in the MAC and PHY layers. UE simulation is one of the most convenient techniques to modify LTE traffic and the GSM/LTE LabKit is very versatile in this respect, due to the YateENB software module.

The GSM/LTE LabKit offers an easy way to operate a controlled LTE network for testing UE devices, including smart phones and embedded IoT/M2M devices. However, one of the limitations of an isolated test network is that it has very little traffic, which limits the scope of the testing and may hide problems in the devices under test.
This is why the GSM/LTE LabKit includes a built-in feature that generates virtual UEs inside the scheduler, resulting in different levels of activity in the MAC and PHY layers of the LTE network. In their turn, such modifications affect the behavior of the real UE devices under test.
Simulated UEs can be active or idle, and they load the resource scheduler and the DL radio channel in the same way as real UEs in the test environment.



1. CREATING THE VIRTUAL UES

First, you have to login into the LabKit, by SSH.

```
ssh yatebts@YOUR_LABKIT_IP -p 54321
```

The password is the serial number printed on the front plate of your GSM/LTE LabKit. Now, you have to use Telnet to access the **rmanager** interface:

```
telnet 0 5037
```

This is where actual UE simulation begins, by means of the **YateENB** module. Using the `enb uepool sim` command, you can create and destroy idle and active virtual UEs:

- **Idle virtual UEs** maintain radio connections (RNTIs) by requesting small amounts of **uplink** bandwidth every few seconds.
- **Active virtual UEs** generate **downlink** traffic at about 100 kbits/sec each, with a low modulation efficiency.

To create N idle virtual UEs:

```
enb uepool sim idle=N
```

To create N active virtual UEs:

```
enb uepool sim active=N
```

To remove all of the virtual UEs:

```
enb uepool sim clear=true
```

2. ANALYZING THE SIMULATED ACTIVITY

The effect of the virtual UEs can be verified and measured inside the eNodeB using the same tools as for real UEs. Throughout all the steps below, **you have to be logged into Telnet.**

2.1. In the Scheduler

The `enb uepool chans` command gives performance information on all of the connected UEs currently known to the scheduler, listed by **RNTI,** including virtual UEs. While simulated UEs **are suffixed by an S** on the RNTI column, idle and active simulated UEs can be distinguished **by the downlink traffic, which is 0 for idle.** For example:

```
enb uepool sim active= 4
added simulated UE's, 4 active, 0 idle
enb uepool sim idle= 10
added simulated UE's, 0 active, 10 idle
enb uepool chans
RNTI    AVGRSSI CURRSSI ...  DL-eff  256QAM  DLbytes DL-TBs  DL-skip DL-NACK ... Age    Idle    UL-BSR  DL-BSR
75       -3      -2     ...  54      false   56k     234     0%      5%      ... 4      0       400     5678
74S     -48     -48     ...  15      false   0       0       0%      0%      ... 5      5       0       0
73S     -49     -49     ...  15      false   0       0       0%      0%      ... 5      5       0       0
72S     -48     -48     ...  15      false   0       0       0%      0%      ... 5      5       0       0
71S     -48     -48     ...  15      false   0       0       0%      0%      ... 5      5       0       0
70S     -48     -48     ...  15      false   0       0       0%      0%      ... 5      5       0       0
69S     -49     -49     ...  15      false   0       0       0%      0%      ... 5      5       0       0
68S     -49     -49     ...  15      false   0       0       0%      0%      ... 5      5       0       0
67S     -49     -49     ...  15      false   0       0       0%      0%      ... 5      5       0       0
66S     -49     -49     ...  15      false   0       0       0%      0%      ... 5      5       0       0
65S     -48     -48     ...  15      false   0       0       0%      0%      ... 5      5       0       0
64S     -48     -48     ...  22      false   22k     284     0%      49%     ... 12     0       0       8108
63S     -49     -49     ...  22      false   41k     516     0%      36%     ... 12     0       0       3744
62S     -48     -48     ...  22      false   145k    1761    0%      55%     ... 12     0       0       1681
61S     -48     -48     ...  22      false   201k    2443    0%      32%     ... 12     0       0       9592
```


In the example above, we have:

- **1 real UE** (RNTI 75)
- **10 idle virtual UEs** (RNTI 65-74)
- **4 active virtual UEs** (RNTI 61-64).

We have edited out PUSCH and PUCCH statistics for better readability.

2.2. In the eNodeB Internal Performance Report

Virtual UEs do not have bearers and do not generate activity above the MAC layer. However, the activity that they produce in the MAC and PHY layers is apparent in KPI-related measurements and in the output of the `enb measurements report` command.

Here is an example with four active virtual UEs in a 5 MHz eNodeB:

```
enb measurements report

Rates are in kbits/sec
In/out counts are in kBytes.

DL Performance Report:
baseline=29 seconds
PDCP:  in=0 rate=0 util=0 loss=0 delay=0ms
RLC:  in=0 rate=0 util=0 rtx/total=0 nacks/pdus=0 nacks/(nacks+acks)=0 overhead=1
MAC:  in=0 rate=0.000269397 util=4.19733e-07 overhead=0.999999 rbUtil=0.59 pdcchCon-
gest=0.47 drop=0.00123564 meanTbs=672 meanBsr=0 bsrTxTime=0ms
HARQ:  in=1329 rate=366.533 util=0.571076 rtx=0.0324608 nack=0.0320923 nackAck=0.482955
renack=0.0249307 conf=0.11 fbRate=0.0647224 unans=0.0488854 pucchRate=0.578199 pus-
chRate=0
PHY:  in=1373 rate=378.83 util=0.590235 avail=641.829 eff=0.230124 eleOverhead=0.32

UL Performance Report:
baseline=29 seconds
PDCP:  out=0 rate=0 util=0 loss=0
RLC:  out=0 rate=0 util=0 nack=0 rtx=0 rej=0
MAC:  out=0 rate=0.000269397 util=3.37592e-07 overhead=0.992958 rbUtil=0.51 drop=0
meanTbs=227 meanBsr=6 delay=174000ms
HARQ:  out=0 rate=0.0382543 util=4.7938e-05 rtx=0.999823 fer=1
PHY:  out=784 rate=216.36 util=0.27113 avail=797.995 eff=0.23843 snr=-10 cap=0 rbOver-
head=0.184
```

In this example, all of the activity in the HARQ and PHY is coming from the virtual UEs. Together, these virtual UEs occupy:

- 59% of the available downlink radio bandwidth
- 51% of the available uplink radio bandwidth
- most of the available DCI bandwidth on PDCCH (PDCCH congestion rate is 47%),

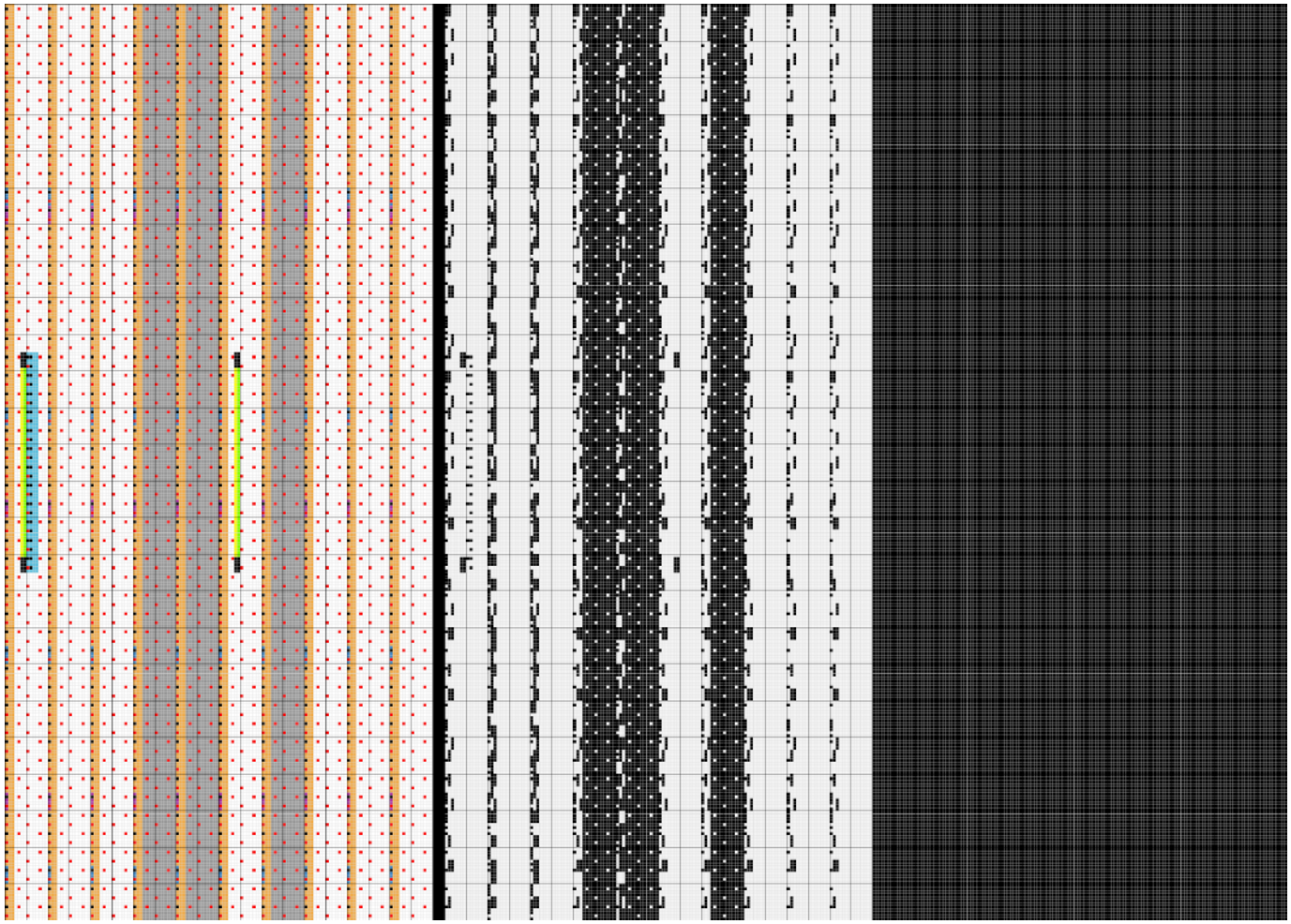
Since these UEs do not really exist, the uplink error rate is 100% and the HARQ feedback rate on PDSCH is very low.

2.3. On the Radio Channel

Virtual UEs produce downlink radio transmissions on PDCCH and PDSCH, just like real UEs.

In order to visualize the radio transmission from virtual UEs in the downlink resource grid and time-frequency power map, you can use the `enb phy printgrid` command, which results in a **.png** file.

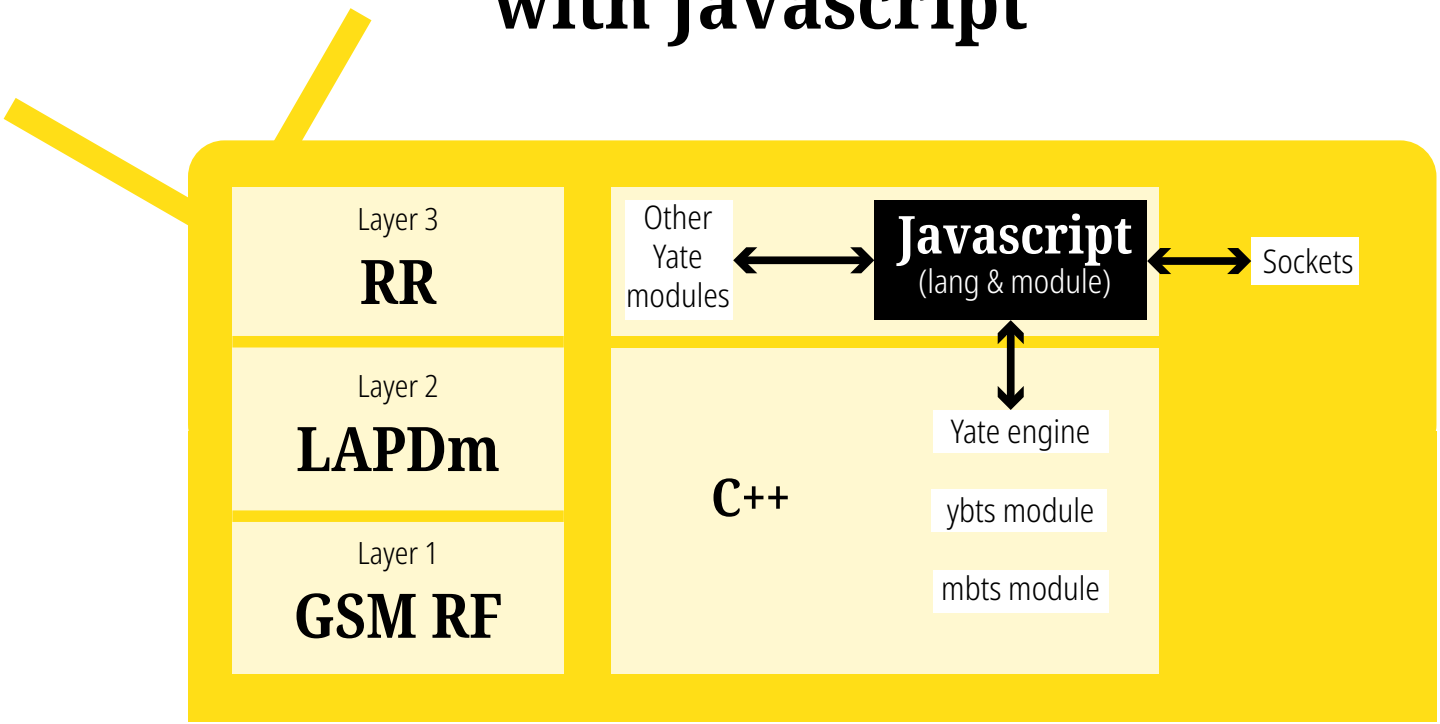
The folllowing example shows the uplink and downlink resource grids for a single 5 MHz LTE frame, with four simulated UEs and no real UEs.



- The **left 1/3** of the image shows the **downlink** grid color-coded by channel type:
  - Green for PSS/SSS.
  - Blue for PBCH.
  - Orange for PDCCH.
  - White for occupied PDSCH; grey for unused PDSCH.
  - Red for CSRS.
  - Black for reserved.
- The **middle 1/3** of the image shows the **downlink grid power levels,** with white bweing maximum power.
- The **right 1/3** of the image shows the **uplink grid power levels;** in the example, it is all black because there is no uplink activity.

APPLICATION NOTE APPN004 2018-02-22

Controlling the GSM/LTE LabKit with Javascript



by ***David A. Burgess***

Due to YateBTS, the GSM/LTE LabKit allows control of GSM/GPRS services such as messages, speech calls, SMS, voice calls, USSD or data traffic by means of Javascript functions. Use of Javascript is very convenient in various laboratory scenarios where networks have to be simulated or radio measurements are necessary. In this Application Note, we are describing the use of Javascript inside YateBTS, and also offering you a series of code examples which are released under a GPL license, so that you can use them as the basis for your various customizations.

1. SCOPE AND AUDIENCE

This note is intended for systems integrators who what to use YateBTS to:

- simulate GSM/GPRS mobile networks in testing environments
- connect YateBTS into simplified roaming gateways
- use YateGSM to make measurements on the GSM radio interface (power, timing advance, etc.).

The level of detail in this note is intended to give technical managers and developers a clear idea of what is possible and some idea of the level of effort required for their own applications.

2. INTRODUCTION

YateBTS is based on the **Yate** telephony application server, which is written in a mix of C++ and Javascript. YateBTS GSM/GPRS functions **up to Layer 2** (LAPDm and RLC) are implemented in C++, but most **Layer 3** functions are implemented in **Javascript**, using an embedded Javascript virtual machine inside Yate.

The Javascript is available to end-users and can be modified or replaced to support custom applications and communicate with outside systems over sockets or using standard protocols already supported by Yate. The various components of Yate/YateBTS communicate among them according to the following flows:

handset →GSM→ ybts+mbts GSM L1/L2 in C++ →messages→ L3 in Javascript →sockets→ custom protocols

handset →GSM→ ybts+mbts GSM L1/L2 in C++ →messages→ L3 in Javascript →other Yate modules→ standard telecom protocols.

3. WHAT YOU CAN DO WITH JAVASCRIPT INSIDE YATEBTS

YateBTS supports the following services:

- registration/authentication
- speech calls
- text messaging (SMS)
- USSD (commercial releases only)
- SMSCB
- GPRS.

The Javascript approach described in this application note allows **complete control** over these services, down to Layer 3, including:

- IMSI/IMEI information
- TMSI information and control
- radio channel information (power, timing).

4. WHY USE JAVASCRIPT?

- The Yate Javascript APIs give complete access to the GSM protocol at Layer 3 and higher, for both **information** and **control**.
- The effort required to write your integration directly in Javascript is probably **less than the effort to write an outside gateway** between your application and some socket-based protocol like SIP
  - by the way, if you really want SIP, there is already Javascript that implements a SIP interface for YateBTS that is given as an example in this application note.

Writing custom applications for YateBTS is a matter of writing Javascript functions that process uplink GSM messages and respond with downlink GSM messages. The following sections give you more technical background on how that works.

5. YATE MESSAGE PASSING

Yate is designed as **a message-passing system.** The structure of Yate is a central **engine** and a collection of **modules** that provide different services, like SIP interfaces, telnet access, call routing, SS7 interfaces, etc. YateBTS is a subset of modules in Yate.

- The modules communicate by passing messages.
- The messages are sets of key-value pairs, as strings.
- Each module can install handlers for different message types with specific priority ordering.

When a module sends out (or “dispatches”) a message, the engine offers the installed handlers for that message type to the message, in a priority order. Priority numbers run **1..100,** with **100 being the strongest priority** and handled first.

Each module can:

- Handle the message and then tell the engine to **delete** it so that no more handlers are called.
- Handle the message, possibly modifying it, and then tell the engine to **continue offering it to handlers** in other modules, continuing in priority order.

Among the Yate modules, ybts translates between GSM L3 messages and Yate internal messages. The Javascript virtual machine **is also a module.** Javascript functions can be defined to handle and dispatch Yate messages. The Javascript functions can handle messages converted from GSM L3 by ybts and also dispatch messages that will be handled by ybts and converted back to GSM L3.

The overall message flow is:

handsets →GSM→ ybts →messages→ Yate engine →messages→ Javascript.

For more complete information about messages, see the following page in the Yate documentation: http://docs.yate.ro/wiki/Messages.

5.1. Example 1: Sending a Yate Message to YateBTS

For example, here is a Javascript function to send an SMS text message from Javascript using a Yate message:

```
// Send text as SMS to the handset with the given IMSI.
// text, IMSI and senderNumber are all strings
function sendSms(text, IMSI, senderNumber)
{
    // All of the fields of the SMS TPDU and RPDU can be controlled here,
    // but this example relies on default values for simplicity.

    // Create the Yate message object.
    var m = new Message("msg.execute");

    // Add the RPDU source and destination numbers (SMSC numbers)
    m.caller = "123456";
    m.called = "123456";

    // Add the TPDU source number (sender mobile number)
    m["sms.caller"] = senderNumber;

    // Add the message body.
    m.text = text;

    // Add the destination IMSI
    m.callto = "ybts/IMSI" + IMSI;

    // Send the message Yate message, which will be handled by YateBTS,
    // resulting in an SMS being sent to the handset with the given IMSI.
    m.dispatch();
}
```

The flow is:

Javascript →message→ Yate engine →message→ ybts →GSM→ handset.

5.2. Example 2: Handling a message

The following example receives and logs all of the mobile-originated SMS text messages sent through YateBTS. It returns false to allow the message to continue to be processed by any other module that might handle it. Had it returned true, the processing for this message would end at this function.

```
// This is a Javascript function to receive and print an SMS text message
```

```
// sent by a handset through YateBTS.
function onMoSMS(msg)
{
    // msg is the Yate message, which contains the SMS sent by the handset.
    if (msg.callto != "smsc_yatebts")
        return false;

    // Extract some fields from the message and print them to the Yate log.
    // These are just a few of what is available, for a simple example.
    var smsInfo = "Source: " + msg.imsi + " ";
    smsInfo = smsInfo + "Dest: " + msg["sms.called"] + " ";
    smsInfo = smsInfo + "Text: " + msg.text;
    engine.debug(engine.DebugInfo,"MO-SMS: " + smsInfo);

    // Do not stop the engine from calling other handlers for this Yate msg.
    // Maybe there are others.
    return false;
}
// Install function onMoSMS as the handler for the msg.execute message type, with
// priority 80.
Message.install(onMoSMS,"msg.execute",80);
```

The flow is:

handset →GSM→ ybts →message→ Yate engine →message→ Javascript.

6. ACCESSING PHY INFORMATION

Every GSM uplink Yate message from ybts includes a field called phy_info that reports:

- **timing advance** and **timing error,** in GSM symbol periods
  - timing advance is the actual current timing advance
  - timing advance error is the timing deviation of the recevied signal from the current timing advance
- **uplink RSSI** in dB relative to the saturation point of the radio
- actual transmitted **uplink power** in dBm, as reported by the handset on SACCH
- **downlink RSSI** at the phone in dBm, as reported by the handset on SACCH
- the Unix system **time** when the corresponding L1 message frame arrived in the radio.

This information is encoded as a string. Here is an example of such a string:

```
TA=3 TE=0.000 UpRSSI=-18 TxPwr=23 DnRSSIdBm=-48 time=1516381399.897
```

Note: *This information is provided for GSM 2G control messages carried over the radio interface on SDCCH and TCH. It is not present for GPRS control messages.*

Note: *One unit of timing advance corresponds to about 550 meters of propagation delay, however, TA cannot be used as a direct measure of distance because of unknown timing advance bias in the handset.*

7. GPRS CONTROL

GPRS connections are initiated with the **call.route** message, just like telephone calls. When the handset requests a GPRS connection, ybts sends a call.route message having a caller value of **sgsn** and **route_type** value of gprs. Further steps in the connection establishment process are signaled with **call.control** messages.

It is possible to intervene in the GPRS connection management process by installing custom Javascript handlers for these messages, at a higher priority than the existing handlers.

8. SMSCB

SMSCB (Short Message Service Cell Broadcast) is a system for broadcasting text messages to handsets. Although the name includes "SMS", the actual service is **completely independent of SMS** and uses different signaling.

To enable SMSCB in YateBTS, set **Control.SMSCB** to **true** or **yes** in the configuration file before starting YateBTS.

The SMSCB feature cannot be enabled or disabled after YateBTS is started.

YateBTS keeps an internal table of active SMSCB messages, which are repeated periodically.

Sending a SMSCB message

To start sending an SMSCB message, send a Yate call.control message to ybts to add it to the table. The fields of the message include entities conform to the **GSM 03.38** and **GSM 03.41** ETSI technical specifications:

- **targetid** = 'ybts'
- **component** = 'ybts'
- **operation** = 'cbadd'
- **dcs** = data coding scheme, see GSM 03.41 9.2.18 and GSM 03.38 5
- **data** = hex string encoding the message content
- **gs** = geographic scope, see GSM 03.41 9.3.2.1
- **code** = message code, see GSM 03.41 9.3.2.1
- **id** = message id, see GSM 03.41 9.3.2.2.

The message **update number** (GSM 03.41 9.3.2.1) will be updated automatically whenever the message is replaced with another **call.control** message.

Stopping from sending SMSCB messages

To stop sending an SMSCB message, send a Yate call.control message to ybts to remove it from the table, identifying the message by the same id tag used when it was added. The fields of the message are:

- **targetid** = 'ybts'
- **component** = 'ybts'
- **operation** = 'cbdel'
- **code** = same message code used to add the mesage
- **id** = same id used to add the message.

These operations can also be performed from the **rmanager** Telnet interface using the `control ybts` command, giving these same message fields on the command line. For example:

```
control ybts cbadd id=2 code=2 gs=1 dcs=1 data=F4F29C9E769F1B
control ybts cbdel id=2 code=2
```

8. GPL JAVASCRIPT FOR YATEBTS

This section indexes some published examples of custom YateBTS applications written in Javascript. **These are provided publicly under a GPLv2 license.** Any of them might be a good starting point for your own custom integration.

8.1. Sending an SMS from the Yate Telnet interface

The **custom_sms.js** script by Null Team, available at http://voip.null.ro/svn/yatebts/trunk/nipc/custom_sms.js, allows an operator to send a text message to a specific IMSI from the Yate **rmanger** Telnet interface.

How it works:

- Messages from rmanager have the type **chan.control**
- The **onControl** function is installed as the **handler** for chan.control messages
- The **onControl** function extracts **parsed tokens** from the rmanager command and inserts them into the fields of a **msg.execute** message.
- The onControl function **dispatches** the msg.execute message, which is then handled by the ybts module to send the text message to the handset on the radio interface.

The overall operation is:

console →telnet→ rmanager module →message→ Javascript →message→ ybts →GSM→ handset delivery.

8.2. Integrating Yate with a VoIP/SMS Gateway

The Null Team scripts available at http://yate.null.ro/svn/bman/scripts/ give an an example of **integrating YateBTS with an outside VoIP/SMS gateway service**, more specifically - Tropo.

- **bman_regist.js** provides **registration and authenticatation of SIMs** using a local subscriber database and automatically assign temporary local-only numbers.
- **bman_route.js routes calls through the Tropo VoIP gateway,** using a sort of "NAT for phone numbers" routing algorithm that allows many subscribers to share a small set of routable numbers.
- **sms_cache_txt.js** - **routes SMS through the Tropo SMS gateway** and provides a simplistic SMSC-like store and forward service.

The message flow is:

handsets →GSM→ ybts →messages→ local SMSC and PBX functions in Javascript →SIP→ Tropo gateway →various→ PSTN/PLMN.

8.3. Running a Local GSM PBX

The standard "Network in a PC" script by Null Team at http://voip.null.ro/svn/yatebts/trunk/nipc/nipc.js ships with YateBTS. It provides the following functions:

- **local subscriber database** for registration and authentication (readUEs, onRegister, onAuth, onUnregister)
- local phone-to-phone **PBX-type calling** (onRoute)
- local **SMS** delivery and store-and-forward (onSms, onIdleAction)
- option to **configure a SIP trunk** to connect to the PSTN for outbound calls (routeOutside)
- rmanager commands to allow **monitoring** and **control** via Telnet (onCommand, onHelp, onComplete)

The operations are:

handsets →GSM→ ybts →messages→ local SMSC and PBX functions in Javascript →SIP→ PSTN

handsets →GSM/GPRS→ ybts →messages→ local GPRS session mgt →TUN→ Internet access.

8.4. Roaming on YateUCN over SIP

YateUCN is an unified core solution intended for LTE and upgrade from GSM/GPRS to LTE. Our partners at SS7ware provide it together with scripts that define its functionality.

The **roaming.js** script by Null Team, available at http://voip.null.ro/svn/yatebts/trunk/roaming/roaming.js and shipped with YateBTS, provides a **SIP interface** to YateUCN, for interconnection to a mobile operator network. It can also can connect to the YateUCN Mini-Core or Hosted Core, which are SS7ware products intended for testing and demonstration.

The services are:

- **Registration over SIP:** like normal SIP channel-response authentication, but with parameters based on GSM location updating procedure. Hash algorithm is **COMP-128** or **Milenage**. Nonce and result lengths match **GSM RAND** and **SRES** lengths
- **MO-** and **MT- speech calls** with SIP: **handset identity** based on IMSI and **GSM 6.01** full rate codec
- **MO-** and **MT- SMS: SIP MESSAGE** method, following the same format as for IMS
- **USSD** over SIP (commercial release only)
- **GPRS**, using either local IP breakout or a GTP-U tunnel.

The message flow is:

handsets →GSM→ ybts →messages→ Roaming.js →SIP→ YateUCN or Mini-Core →SS7/SIP→ HLR and/or mobile operators.

or

handsets →GSM→ ybts →messages→ Roaming.js →SIP→ Hosted Core.

The GSM/LTE LabKit

The **GSM/LTE LabKit** is a highly configurable and versatile 2,5G/4G BTS/eNodeB, which is intended for laboratory use/applications development. The LabKit is built on general purpose hardware, starting from an Intel main board, with software-defined radio, and runs **YateBTS** on a Mageia Linux OS. It has four working modes, including LTE and Network in a PC (NiPC). While the NiPC mode provides the full functionality of a GSM network, the other GSM and LTE modes need a hosted core/minicore in order to achieve full functionality, which is also provided to LabKit clients.
The LabKit is easily configurable both locally and remotely, by SSH and HTTP interfaces. One of its diferentiating features resides in the radio traffic monitoring functionalities, which are well documented in a series of Yate/YateBTS App Notes.
The online version of the LabKit public documentation can be consulted at https://wiki.yatebts.com/index.php/Lab_Kit.
The LabKit comes with an affordable price tag, **$6,000.**

Yate and YateBTS software and related hardware products are own and operated by a group of American-Romaniain companies including Null Team, SS7ware and Legba.

CONTACTS

Sales: **Lucian Bîlă,** lucian@ss7ware
www.yate.ro, www.yatebts.com
www.twitter.com/yate_VoIP
Romania office: Str. General Praporgescu 1-5, et. 5/6, ap. 10, 11, 14, Sector 2, 020965, Bucharest, Romania.

