UNIVERSIDAD AUTONOMA DE COAHUILA

FACULTAD DE SISTEMAS

A GSM Base Transceiver Station: Technical manual

Materia: Comunicaciones II

Catedrático: M.I. Francisco Gerardo Hernandez Rivera

Alumnos: Alfredo Valdés Cárdenas

Abraham Antonio Iracheta Rodríguez

Jeny Abigail Mendoza Zapata

Adrián Daniel Flores Rivera

23 de Noviembre de 2015

Set Up

Components

Linux Server

The first requirement is a standard commodity Linux server. Other architectures are beginning to be supported, but stick to an x86 processor running a 32-bit operating system for the best results for now. Minimum requirements for processing power and RAM are not clearly defined due to the many variables involved, such as the number of concurrent carrier signals, network load, network usage type, radio environment, etc.

Software Defined Radio

The software defined radio (SDR) is the key breakthrough that makes OpenBTS possible from a hardware perspective. A modern SDR is a piece of hardware that connects to your computer via a USB cable or over Ethernet. The SDR hardware implements a completely generic radio that can send and receive raw waveforms in a defined frequency range (i.e., 60 MHz to 4 GHz) with a host application. That host application could be an FM radio implementation that receives the raw waveforms, demodulates the signal, and plays back the audio.

Antennas

Many SDRs have enough transmit and receive sensitivity to operate without antennas in a small environment. Typically, a coverage area with a radius of 1 m can be achieved. The coverage areas will not overlap and interfere with each other. The frequency also plays a role in the coverage area size. Low-frequency bands (850 and 900 MHz) propagate larger distances than high-frequency bands—sometimes nearly twice as far.

Test Phones

For testing, at least two GSM handsets compatible with the band you will be using (850, 900, 1800, or 1900 MHz) are needed.

Operating System and Development Environment Setup

OpenBTS has traditionally been developed and tested on Ubuntu Long-Term Support (LTS) distributions. It has also been tested on Debian and CentOS distributions. Preliminary packaging for RPM-based systems (CentOS, Fedora, and Red Hat Enterprise Linux) is also available.

Git Compatibility

Git is a version-control system that manages software source code changes. The OpenBTS project utilizes several new features in Git, such as submodule branch tracking. First, execute this command to add support for Personal Package Archives, an alternate way to distribute binary release packages:

$ sudo apt-get install software-properties-common python-software-properties

Then, execute the following command to add a repository for the latest Git builds to your system:

$ sudo add-apt-repository ppa:git-core/ppa

Now, you must simply refresh the list of packages and install Git again to update your system's client:

$ sudo apt-get update $ sudo apt-get install git

To confirm that the new Git client is installed properly, run the following command:

$ git --version git version 1.9.1

Downloading the Code

The OpenBTS project consists of multiple software components hosted in separate development repositories on GitHub. To download these development scripts into your new environment, run the following command:

$ git clone https://github.com/RangeNetworks/dev.git

Now, to download all of the components, simply run the clone.sh script:

$ cd dev
$ ./clone.sh

Now that the OpenBTS project sources are in your development environment, you can select a specific branch or release to compile. For example, if you wanted to build the v4.0.0 release, run the following command:

$ ./switchto.sh v4.0.0

The current version target can be listed for each component by using the state.sh script.

$ ./state.sh

Building the Code

To compile binary packages, we will use the **build.sh script.** It automatically installs the compiler and auto configuration tools as well as any required dependencies. It also controls which radio transceiver application will be built. As there are several different drivers available for the various radio types, build.sh requires an argument so it knows which hardware is being targeted (valid radio types are SDR1, USRP1, B100, B110, B200, B210, N200, and N210).

Run the build command now:

$ ./build.sh N210

When the build script finishes, you will have a new directory named "BUILDS" containing a subdirectory with the build's timestamp. An example listing of this directory follows:

$ ls dev/BUILDS/2015-11-04--20-44-51/*.deb
liba53_0.1_i386.deb **range-asterisk-config_5.0_all.deb**
libcoredumper1_1.2.1-1_i386.deb **range-configs_5.0_all.deb**
libcoredumper-dev_1.2.1-1_i386.deb **sipauthserve_5.0_i386.deb**
openbts_5.0_i386.deb **smqueue_5.0_i386.deb**
range-asterisk_11.7.0.4_i386.deb

Change into your new build directory before continuing:

$ cd dev/BUILDS/2015-11-04--20-44-51/

Installing Dependencies

We'll need to install some additional system libraries and define an additional repository source so all dependencies can be found and installed.

Execute the following commands to define an additional repository source for ZeroMQ, a library that all the components use:

$ sudo apt-get install software-properties-common python-software-properties
$ sudo add-apt-repository ppa:chris-lea/zeromq
$ sudo apt-get update

Coredumper library

OpenBTS uses the coredumper shared library to produce meaningful debugging in- formation if OpenBTS crashes. libcoredumper-dev contains development files needed to compile programs that utilize the coredumper library, and libcoredumper contains the shared library that applications load at runtime:

$ sudo dpkg -i libcoredumper1_1.2.1-1_i386.deb

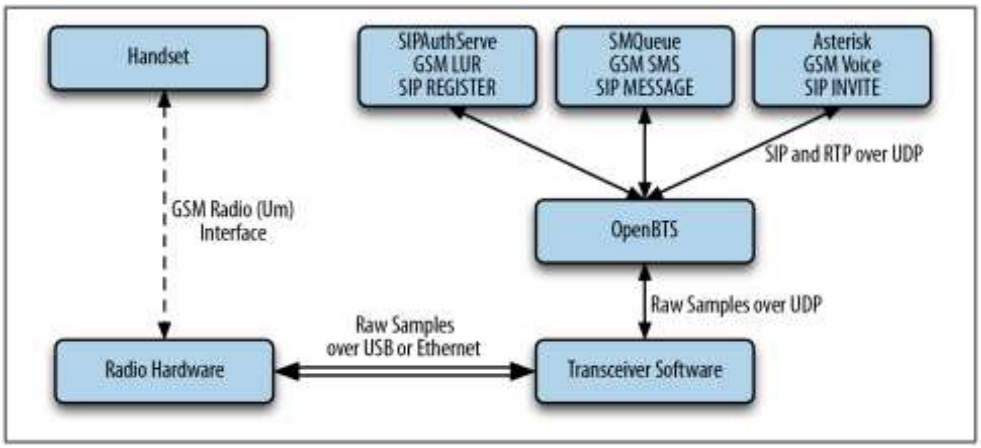
A5/3 library

OpenBTS uses the A5/3 shared library to support call encryption. It contains crypto- graphic routines that must be distributed separately from OpenBTS:

$ sudo dpkg -i liba53_0.1_i386.deb

Installing Components

By installing all of the following components on a fresh system, you are guaranteed a functional GSM network-in-a-box. Everything needed for voice, SMS, and data will be running in a single system.



System configs

This package contains a set of default configurations that will allow a fresh Ubuntu system to work out of the box when installed. It includes settings for the network in- terface, firewall rules, domain name system (DNS) configuration, logging, etc.

$ sudo dpkg -i range-configs_5.0_all.deb

Asterisk

Asterisk is a VoIP switch responsible for handling SIP INVITE requests, establishing the individual legs of the call, and connecting them together.

$ sudo dpkg -i range-asterisk*.deb
$ sudo apt-get install –f

SIPAuthServe

SIP Authorization Server (SIPAuthServe) is an application that processes SIP REGIS- TER requests that OpenBTS generates when a handset attempts to join the mobile net- work. IPAuthServe is responsible for

updating the subscriber registry database with the IP address of the OpenBTS instance that initiated it, allowing other subscribers to call the handset:

$ sudo dpkg -i sipauthserve_5.0_i386.deb
$ sudo apt-get install –f

SMQueue

SIP MESSAGE Queue (SMQueue) is an application that processes SIP MESSAGE re- quests that OpenBTS generates when a handset sends an SMS.

$ sudo dpkg -i smqueue_5.0_i386.deb
$ sudo apt-get install -f

OpenBTS

OpenBTS is responsible for implementing the GSM air interface in software and communicating directly with GSM handsets over it.

$ sudo dpkg -i openbts_5.0_i386.deb
$ sudo apt-get install –f

Starting/Stopping Components

Now that each component has been installed, you need to start them. Components are controlled on Ubuntu with a system named Upstart. To start all components, execute the following:

$ sudo start asterisk
$ sudo start sipauthserve
$ sudo start smqueue
$ sudo start openbts

Conversely, to stop all components, use:

$ sudo stop openbts
$ sudo stop asterisk
$ sudo stop sipauthserve
$ sudo stop smqueue

Initial State

Some of the manual steps that follow will conflict and fail if other instances of the services are already running. To make sure that nothing else is running on this system, execute the following:

```
$ sudo stop openbts
$ sudo stop asterisk
$ sudo stop sipauthserve
$ sudo stop smqueue
```

Now you can proceed to confirm connectivity at each step in the chain before running the first basic tests.

The default IP address for all Ettus hardware is 192.168.10.2. Assign an appropriate address to your Ethernet interface using the following example:

```
$ sudo ifconfig eth1 192.168.10.1/24
```

Now test the connection with a simple ping. Press Ctrl-C to stop the ping test:

```
$ ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_req=1 ttl=64 time=1.037 ms
64 bytes from 192.168.10.2: icmp_req=2 ttl=64 time=1.113 ms
^C
```

Now to test that the UHD driver recognizes the USRP we execute:

```
$ uhd_usrp_probe
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.007.002-release
-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes
_____________________________________________________
/
| Device: USRP2 / N-Series Device
| _____________________________________________________
| /
| | Mboard: N210r4
| | hardware: 2576
| | mac-addr: XX:XX:XX:XX:XX:XX
| | ip-addr: 192.168.10.2
| | subnet: 255.255.255.255
| | gateway: 255.255.255.255
| | gpsdo: none
| | serial: XXXXXX
```

```
| | FW Version: 12.3
| | FPGA Version: 10.0
| |
| | Time sources: none, external, _external_, mimo
| | Clock sources: internal, external, mimo
| | Sensors: mimo_locked, ref_locked
| | _____________________________________________________
| | /
| | | RX DSP: 0
| | | Freq range: -50.000 to 50.000 Mhz
| | _____________________________________________________
| | /
| | | RX DSP: 1
| | | Freq range: -50.000 to 50.000 Mhz
| | _____________________________________________________
| | /
| | | RX Dboard: A
| | | ID: SBX (0x0054)
| | | Serial: XXXXXX
| | | _____________________________________________________
| | | /
| | | | RX Frontend: 0
| | | | Name: SBXv3 RX
| | | | Antennas: TX/RX, RX2, CAL
| | | | Sensors: lo_locked
| | | | Freq range: 400.000 to 4400.000 Mhz
| | | | Gain range PGA0: 0.0 to 31.5 step 0.5 dB
| | | | Connection Type: IQ
| | | | Uses LO offset: No
| | | _____________________________________________________
| | | /
| | | | RX Codec: A
| | | | Name: ads62p44
| | | | Gain range digital: 0.0 to 6.0 step 0.5 dB
| | | | Gain range fine: 0.0 to 0.5 step 0.1 dB
| | _____________________________________________________
| | /
| | | TX DSP: 0
| | | Freq range: -250.000 to 250.000 Mhz
| | _____________________________________________________
| | /
| | | TX Dboard: A
```

```
| | | ID: SBX (0x0055)
| | | Serial: XXXXXX
| | | ____________________________________________________
| | | /
| | | | TX Frontend: 0
| | | | Name: SBXv3 TX
| | | | Antennas: TX/RX, CAL
| | | | Sensors: lo_locked
| | | | Freq range: 400.000 to 4400.000 Mhz
| | | | Gain range PGA0: 0.0 to 31.5 step 0.5 dB
| | | | Connection Type: QI
| | | | Uses LO offset: No
| | | ____________________________________________________
| | | /
| | | | TX Codec: A
| | | | Name: ad9777
| | | | Gain Elements: None
```

Confirm Radio Connectivity

The first thing we're going to verify is that the transceiver application can communicate with the radio hardware.

```
$ cd /OpenBTS
$ sudo ./transceiver
[sudo] password for openbts:
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.006.002-release
Using internal clock reference
-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes
```

In case that we see the following output

```
$sudo ./transceiver
linux; GNU C++ version 4.8.2; Boost_105400; UHD_003.008.000-release
Using internal clock reference
-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes
UHD Warning:
   The recv buffer could not be resized sufficiently.
   Target sock buff size: 50000000 bytes.
```

Actual sock buff size: 212992 bytes.
See the transport application notes on buffer resizing.
UHD Warning:
The recv buffer could not be resized sufficiently.
Target sock buff size: 50000000 bytes.
Actual sock buff size: 212992 bytes.
See the transport application notes on buffer resizing.
Please run: sudo sysctl -w net.core.rmem_max=50000000
UHD Warning:
The send buffer could not be resized sufficiently.
Target sock buff size: 1048576 bytes.
Actual sock buff size: 212992 bytes.
See the transport application notes on buffer resizing.
Please run: sudo sysctl -w net.core.wmem_max=1048576

Then we should run the following commands:

$sudo sysctl -w net.core.rmem_max=50000000
$sudo sysctl -w net.core.wmem_max=1048576

Now all should be ready to go.

The Configuration System and CLI

All configuration of OpenBTS is accomplished by manipulating keys stored in an SQLite3 database. By default, this database is stored at **/etc/OpenBTS/OpenBTS.db**. Each key is defined in a schema that is compiled in OpenBTS and used to validate the values being used. One advantage afforded by this style of configuration system is that most key values can be changed and are applied to the running system within a few seconds without interrupting service. These are dynamic keys. There are also a few static keys in the configuration system that require a restart of OpenBTS to apply the change.

The easiest way to manipulate the configuration keys is via the OpenBTS commandline interface (CLI). Run the following shell command to open it:

$ sudo /OpenBTS/OpenBTSCLI

You are now presented with an OpenBTS prompt. This is where commands, including configuration changes, can be executed for processing by OpenBTS. From now on, commands prefixed with $ are to be executed on the Linux command line. Commands prefixed with OpenBTS> are for the OpenBTS command line. It may be convenient to have two terminal windows open so there is no need to constantly enter and exit the OpenBTS command line.

Changing the Band and ARFCN

The first things we must check are the radio band and Absolute Radio Frequency Channel Number (ARFCN) being used. The radio band is one of four values: 850, 900, 1800, or 1900 MHz, corresponding to the four GSM bands available around the world.

An ARFCN is simply a pair of frequencies within the selected band that will be used for the transmission and reception of radio signals. Each radio band has over 100 different ARFCNs that can be used. ARFCN may also be referred to as the carrier (e.g., systems using multiple ARFCNs are multiple carrier systems). Choosing the correct band and ARFCN is important for regulatory reasons and to avoid interference with or from local carriers. You use the OpenBTS config command to inspect the current band and ARFCN settings. These configuration keys are in the GSM.Radio category. To view all configuration keys with the word GSM.Radio in their name, enter the following command:

OpenBTS> config GSM.Radio
GSM.Radio.ARFCNs 1 [default]
GSM.Radio.Band 900 [default]
GSM.Radio.C0 51 [default]
GSM.Radio.MaxExpectedDelaySpread 4 [default]
GSM.Radio.PowerManager.MaxAttenDB 10 [default]
GSM.Radio.PowerManager.MinAttenDB 0 [default]
GSM.Radio.RSSITarget -50 [default]
GSM.Radio.SNRTarget 10 [default]

The GSM.Radio.Band key shows that the 900 MHz band is being used and the GSM.Radio.C0 key indicates that ARFCN #51 in that band is currently selected. We shall leave it as is to experience less interference from the other bands, given that in México we use the 850 and 1900 MHz bands.

Shortname

The shortname is displayed on some handsets when browsing. It's the first thing someone will notice when searching for the network, so go ahead and change it from the default of OpenBTS:

OpenBTS> config GSM.Identity.ShortName PumaBTS

GSM.Identity.ShortName changed from "OpenBTS" to "PumaBTS"

Searching for the Network

Now that the radio is calibrated and the settings are confirmed, we will use a handset to search for the newly created network. Each handset's menu is different but the item is usually similar to "Carrier Selection" or "Network Selection."

- Launch the "Settings" application from the Android menu system.
- Select "More."

- Select “Mobile networks.”
- Select “Network operators.” This may or may not start a search. If it does not, select “Search networks.”
- Once the search has finished, a list of available carrier networks is presented.

If we were to use an iPhone we should do as it follows:

- From the home screen, open the “Settings” app.
- Select “Carrier.”
- On the “Network Selection” screen, disable the automatic carrier selection.
- The handset will now search for available carrier networks.
- Once the search has finished, a list of available carrier networks is presented.

Here we see the test network in the list of selectable carriers. Depending on the handset model, firmware, and SIM used, the network ID will be displayed as “00101,” “001-01,” “Test PLMN 1-1,” or the GSM short name of “PumaBTS.” If your test network is not detected, force the search again by either reselecting the menu item, toggling airplane mode between on and off, or power cycling the handset. If that still does not work, confirm again that the handset supports the GSM band you have configured above and that the baseband is unlocked (i.e., not restricted by contract to only using a specific carrier).

The only step left before actually connecting to your test network is to find and enter your handset’s identity parameters so it will be accepted onto the network.

Finding the IMSI

The main identity parameter you will be searching for is the International Mobile Subscriber Identity (IMSI). This is a 14–15 digit number stored in the SIM card and is analogous to the handset’s username on the network. Handsets will not usually divulge the IMSI of their SIM card. It can sometimes be located in a menu or through a field test mode, but this method of determining a SIM’s IMSI is very cumbersome to explain. Luckily, there are other methods; OpenBTS also knows the IMSIs it has interacted with and, because you are in control of the network side, you also have access to this information.

To force an interaction between a handset and your test network, you will perform a location update request (LUR) operation on the network, analogous to a registration. This is nothing more complicated than selecting the network from the carrier selection list.

Before attempting any LURs, you need to start the SIPAuthServe daemon responsible for processing these requests:

```
$ sudo start sipauthserve
sipauthserve start/running, process 7017
```

OpenBTS remembers these LUR interactions in order to perform something called IMSI/Temporary Mobile Subscriber Identity (TMSI) exchanges. IMSI/TMSI exchanges swap the user-identifiable IMSI for a TMSI and are used to increase user privacy on the network. The exchanges are disabled by default (modify

Control.LUR.SendTMSIs to enable); however, the information is still there to inspect using the tmsis command.

Use it now to view all recent LUR interactions with handsets:

```
OpenBTS> tmsis
IMSI            TMSI  IMEI            AUTH  CREATE  ACCESSED  TMSI_ASSIGNED
334020256132746 -     012546629231850 0     78s     78s       0
```

Entries are sorted by time, with the top entries corresponding to the most recent interactions. Your handset should be the top entry on this list—the most recent interaction with AUTH set to 0 because the LUR failed due to the handset not being a known subscriber; When we manage to sign up a subscriber this number will change to 1.

Finding the IMEI

In a busy environment, it can be difficult to ascertain which handset hardware corresponds to which entry on this list. To match an IMSI to a specific piece of hardware, you can use the International Mobile Equipment Identifier (IMEI). It is the unique identifier given to the handset's physical radio hardware, analogous to a MAC address on an Ethernet interface.

A handset's IMSI is usually printed under its battery cover or somewhere very near the SIM itself. On many handsets, the IMEI can also be accessed by dialing the following on the keypad:

***#06#**

The IMEI value is typically only used for reporting and detecting stolen hardware in production environments. Here it serves as a convenient way to determine which SIM is in which handset. The final digit of your IMEI may not match what OpenBTS displays. It is a check digit and is shown as a zero in OpenBTS.

Adding a Subscriber

We should now have all the necessary pieces of information to create a new subscriber account on our test network. A couple of fields are still needed but are freely selectable: Name and Mobile Station International Subscriber Directory Number (MSISDN). The Name field is merely a friendly name for this subscriber so you can remember which handset or which person it is associated with. The MSISDN field is nothing more complicated than the subscriber's phone number. Because you are not connected to the public telephone network, this can be any number you choose. The program you need to add subscribers is nmcli.py. It is a simple client for the NodeManager APIs and allows you to change configuration parameters, add subscribers, monitor activity, etc., all via JSON formatted commands. nmcli.py is already present in your development directory—move there now to access it:

$ cd dev/NodeManager

There are two ways to add a subscriber using nmcli.py. The first creates a subscriber that will use cached authentication:

$ ./nmcli.py sipauthserve subscribers create name imsi msisdn

The second creates a subscriber that will use full authentication:

$ ./nmcli.py sipauthserve subscribers create name imsi msisdn ki

Since we are using a spare SIM from another provider, we do not have access to Ki and should use the first invocation style.

```
$ ./nmcli.py sipauthserve subscribers create "alfredo" IMSI334020256132746 \66611071992
raw request: {"command":"subscribers","action":"create","fields":{"name":"iPhone
4",imsi":"IMSI334020256132746","msisdn":"66611071992","ki":""}}
raw response: {
"code" : 200,
"data" : "both ok"
}
```

Now when we select our test network in the connection menu, the LUR should succeed. This can be confirmed with the tmsis command in OpenBTS. The “AUTH” column will now have a “1” in the entry corresponding to our IMSI:

```
OpenBTS> tmsis
IMSI              TMSI  IMEI            AUTH  CREATED  ACCESSED  TMSI_ASSIGNED
334020256132746   -     012546629231850 1     11m      56s       0
```

We can use the process above to register as many devices as we like.

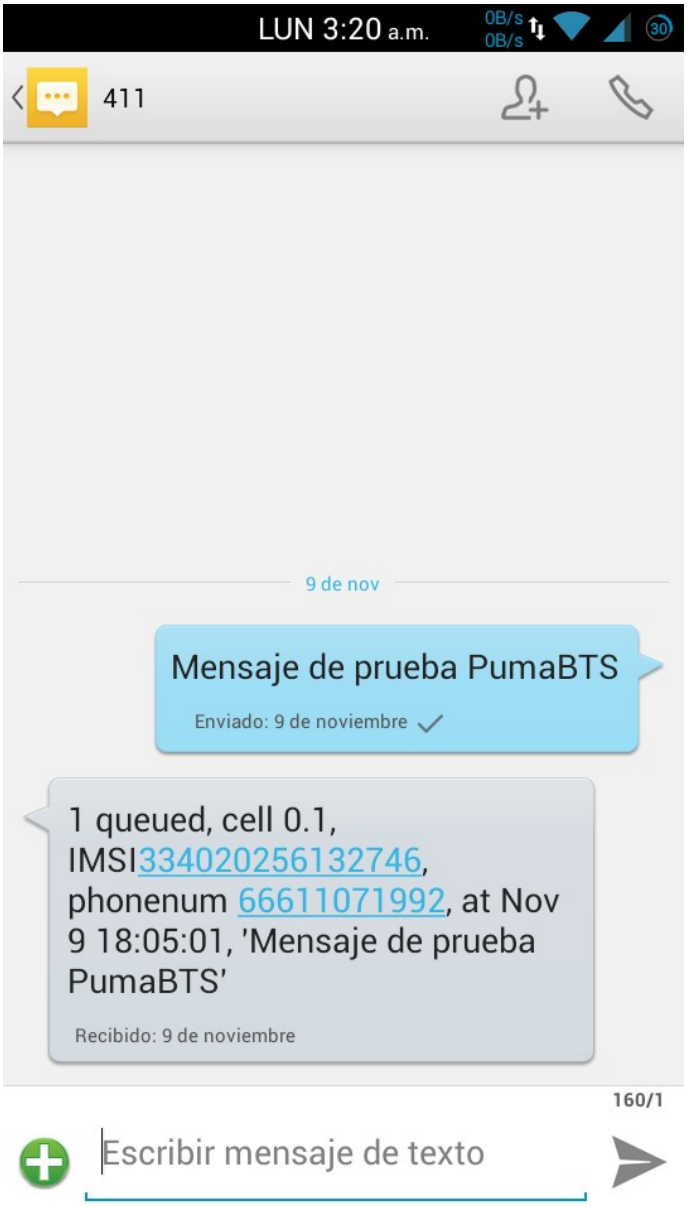
Test SMS

Now that a handset has access to your network, you can perform some more interesting tests. The first is a quick test of your network’s SMS capabilities. The component responsible for receiving, routing, and scheduling the delivery of SMS messages is SMQueue. It must be started before testing out these features; execute the following command to do so:

```
$ sudo start smqueue
smqueue start/running, process 21101
```

On your handset, compose an SMS to the number 411. This is a “shortcode” handler in SMQueue that will simply echo back whatever it receives along with some additional information about the network and subscriber account that was used. The body of the message to 411 can be whatever you’d like, although it can be useful to use unique content for each message or sequential numbers or letters. This helps you

pinpoint which message is being responded to in case an error occurs. Once you have your message composed to 411, hit send. After a few seconds, a reply should appear (an example follows):



This indicates the following:

• There is one message queued for delivery.

• The base station has a load factor of 0.1.

• The message was received from IMSI 334020256132746, MSISDN 66611071992.

• The message was sent on November 9 at 18:05:01.

• The message body was “Mensaje de prueba PumaBTS.”

SMS messages can also be tested directly from OpenBTS by using the sendsms command. From the OpenBTS CLI, let's see how it is invoked by using the help command:

OpenBTS> help sendsms
sendsms IMSI src# message... -- send direct SMS to IMSI on this BTS, addressed from source number src#.

Messages are sent by specifying a target IMSI, the source number the message should appear to have originated from, and the message body itself. Substitute the information for your subscriber account to compose a message and press Enter:

OpenBTS> sendsms 334020256132746 8675309 direct SMS test
message submitted for delivery

After a few seconds, your handset should display a new incoming message from the imaginary number 8675309 with a body of "direct SMS test." SMS messages created in this way do not route through SMQueue at all; they are sent directly out through the GSM air interface to the handset and, as such, cannot be rescheduled. If the handset is offline or unreachable, these messages are simply lost. This is why SMQueue is needed—to attempt and reschedule deliveries in the inherently unpredictable wireless environment. If you have configured more than one handset for use in your network, feel free to send a few messages back and forth between them. Verify that the source numbers are correct when receiving messages and that replies to these messages are routed back to the original sender.

Test Calls

The other service to test is voice. As with SMS, OpenBTS does not directly handle voice and requires an additional service to be run—in this case, Asterisk. Start Asterisk now:

$ sudo start asterisk
asterisk start/running, process 1809

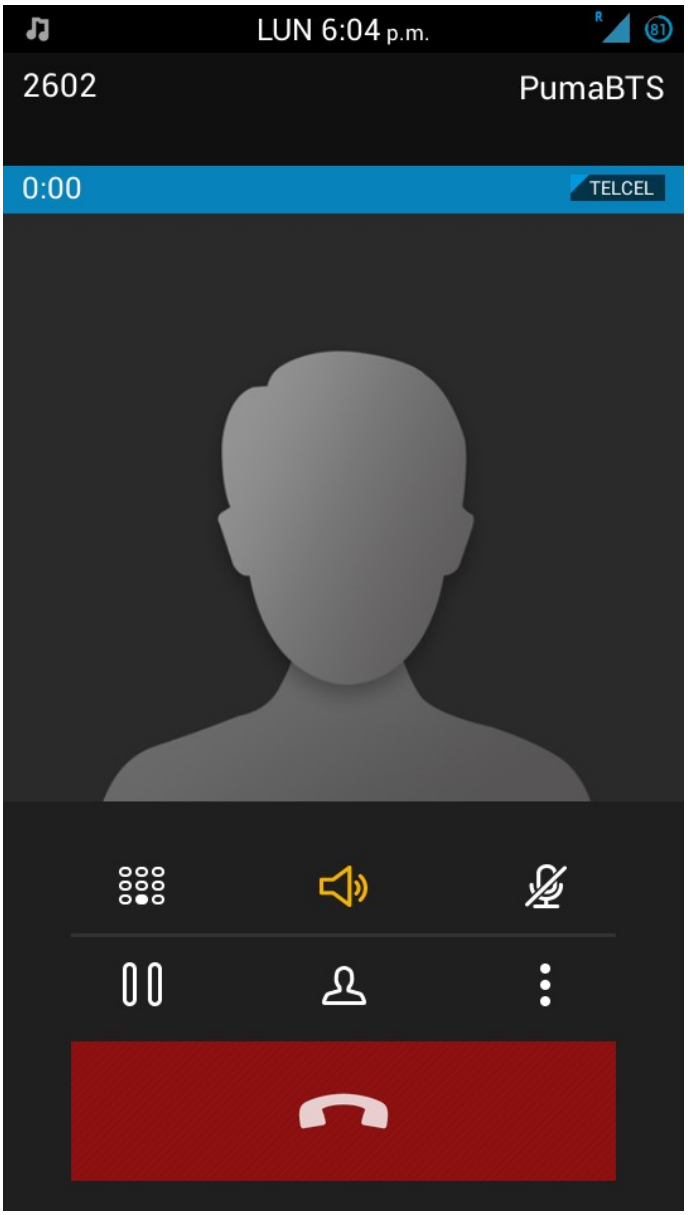
Using the same handset you used in the SMS tests, you will now verify a few aspects of the voice service. This is accomplished by utilizing a few test extensions that the rangeasterisk-configs package defines. An extension is an internal phone number, unreachable from the outside.

Test Tone Call (2602)

The first test extension you will use plays back a constant tone. This might not sound too exciting but does confirm many things about the network:

- Asterisk is running and reachable.
- Call routing is working as expected.
- Downlink audio is functional.

Call 2602 with your handset now. As you listen to the tone, listen for changes in pitch. These changes in pitch are due to missing information in the downlink voice stream path, similar to packet loss. In the field, this is the primary use for the test tone extension: testing downlink quality. A downlink loss of 3% is normal in production networks, with losses of 5%–7% still providing an understandable conversation.



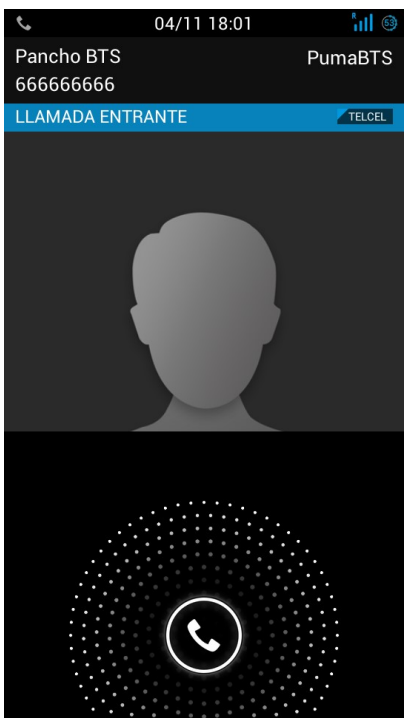
Echo Call (2600)

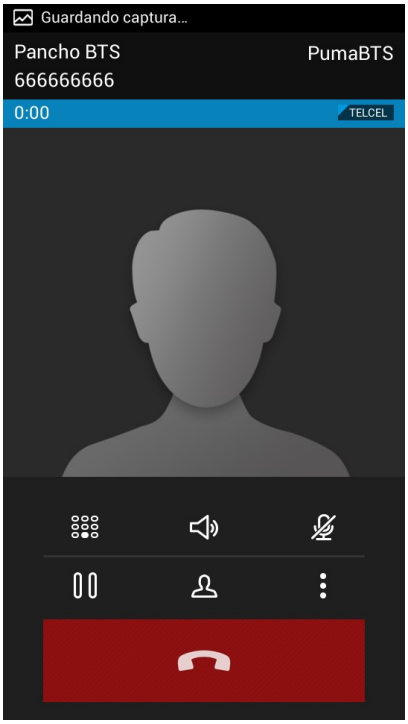
The next test extension creates an “echo call.” Basically, all audio that Asterisk receives will be immediately echoed back to the sender. In addition to confirming the items listed for the test tone call, the echo call will reveal any delay or uplink quality issues present in your network.

Call 2600 with your handset now. As you speak into the microphone, you should hear yourself very shortly afterward in the earpiece. A little delay is normal, but longer delays lead to an experience more like using a walkie-talkie. The human brain can deal with delays up to about 200 ms without trouble. Beyond that, the conversation starts to break down and both sides stop speaking because it becomes uncomfortable.

Two-Party Call

If you have configured more than one handset for use in your network, feel free to place some calls between them. Verify that the source numbers are correct when receiving a call.





GPRS

Mobile networks have, in many areas of the world, been reduced to being data networks. Over-the-top (OTT) services like WhatsApp and Skype only need a data pipe and participants can connect regardless of carrier. Fees between the participants are also not dependent upon geographic location, unlike local versus long-distance charges. GPRS is much too slow to support bidirectional streaming video but can suffice for a low-quality voice call. Its speeds are ideal for email and OTT text messaging. The world of sensors and infrastructure such as heat and flow sensors or electrical and parking meters also needs data connectivity. These low-bandwidth machine-tomachine (M2M) devices, now referred to as Internet of Things (IoT) devices, are a very common use for GPRS. GPRS is actually not a part of GSM. It was developed after GSM had been standardized and is usually referred to as 2.5G, whereas plain GSM is 2G. OpenBTS abstracts these differences and presents a unified configuration where possible.

Enabling/Disabling

By default, the GPRS service is disabled in OpenBTS. Turn it on now by toggling the GPRS.Enable key:

OpenBTS> config GPRS.Enable 1
GPRS.Enable changed from "0" to "1"
GPRS.Enable is static; change takes effect on restart

Restart OpenBTS to apply this static key:

$ sudo stop openbts
$ sudo start openbts

Once OpenBTS has restarted, log back in to its command line and use the gprs list command to confirm that OpenBTS has set up a few channels for GPRS:

OpenBTS> gprs list
PDCH ARFCN=166 TN=1 FER=0%
PDCH ARFCN=166 TN=2 FER=0%

Central Services

GPRS does not rely on any additional components but some configuration must be in place on your Linux host for things to work correctly. This should be taken care of already from the range-configs package during setup but this is how to double-check that things are in order. The handsets' IP traffic is piped through OpenBTS and into a virtual network interface named sgsntun. You can confirm now that OpenBTS has created it by using ifconfig:

$ ifconfig sgsntun
sgsntun Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

The virtual network interface also needs routes and rules applied to it for the iptables Linux firewall. Example rules are located in /etc/OpenBTS/iptables.rules and can be modified if needed to change the gateway interface name. By default, they are written for eth0. Apply the rules now manually:

$ sudo iptables-restore < /etc/OpenBTS/iptables.rules

To have the system apply these rules every time your eth0 interface comes up, modify /etc/network/interfaces to add the final line below, which contains “pre-up”:

 # This file describes the network interfaces available on your system
 # and how to activate them. For more information, see interfaces(5).
 # The loopback network interface
auto lo
iface lo inet loopback
 # The primary network interface
auto eth0
iface eth0 inet dhcp
pre-up iptables-restore < /etc/OpenBTS/iptables.rules

With the tunnel device present and the rules applied, your Linux host should be in order.