S3

Introduction

Welcome to S3, a smaller, simpler simulator.

S3 is a stand alone tool that emulates a subset of an LTE network. By running one or more instances of S3, an LTE network of arbitrary size, load and complexity can be built quickly and easily. S3 ships as a Java jar with some configuration files which runs on any available hardware that supports Java from a raspberry Pi to the latest blades. It has minimal storage requirements as even its log files could be directed to /dev/null if not required. It has a secondary feature in that it can generate and manage the topology of the network as would be reported by an OSS.

S3 emulates the network by producing the CTR events that would be produced by a cell (i.e. with consistent and coherent identifiers and values) in response to calls conforming to the specified patterns. It can also produce corresponding CTUM events that can then be used for IMSI enrichment if required. Events are streamed in real time to the specified destinations exactly as they would be from the nodes.

Key concepts:

S3 simulates calls on a network (more specifically, it generates the events that would result from the interaction of eNodeB and a UE) and since every call is just a pattern of events, it is easy to define the patterns to be emulated and their relative frequency. S3 maintains a pool of the nodes that can make calls, the cells attached to those nodes and the network destinations that events can be sent to. It is worth noting that since S3 simulates calls, it makes little difference to it whether they are being generated from twenty or twenty thousand nodes.

Internally, S3 maintains a circular queue with an entry for each time slot (typically every 125 milliseconds). It then runs two threads, one creates calls and adds the generated events into the appropriate time slots and the other thread transmits all the events that have been generated for a given timeslot. Among other benefits, this allows S3 to self regulate by keeping track of how long each thread needs to do its job and adjusting the generation rate up or down as needed.

To generate a call, S3 takes the next pattern to be processed and creates an instance of each event in the pattern. It selects or generates the key identifiers that all events of that call will share (cellId, MME, RBS, GUMMEI, ENBS1APID, RAC_UE_REF and so on). The use of resource pools makes it easy to ensure that resources such as IMSI's will be exclusive to a given call and can not be re-used until the last event of that call has been transmitted. Since each call may have a random duration, S3 will insert any required periodic events at the times and offsets specified by the pattern. Other then the 'header' identifiers, S3 only sets the parameters for each event that have been specified in the pattern because there is no point setting values nobody is interested in reading. Advanced options exist to allow the operator to tell S3 what proportion of events should use invalid GUMMEIs or corrupt values for ENBS1APID and so on. Having generated all the events for a given call, they are then added to the queues for required timeslots so they can be distributed at the right time.

The distribution thread takes all the events for a specific timeslot, sets the current timestamp and transmits them to the appropriate destinations. Although 'Events per Second' is a common metric for testing, the actual number of events that will be transmitted in any given time period is variable and unpredictable. It can be estimated from the average number of events per call multiplied by the number of calls per second but since the number of events per call is completely dependent on the patterns under test, the primary tool used to control the load that S3 generates is the NumCallsPerSec parameter.

Installation:

Download latest **S3-X.X.X-bundle.zip** from http://eselivm2v238l.lmera.ericsson.se:8081/nexus/content/repositories/assure-releases/com/ericsson/oss/services/S3/

Unzip **S3-X.X.X-bundle.zip** somewhere you can find it on a machine with Java 1.7 or later installed.

Configuration files will be in a sub-directory called "etc", logs files will be in a sub-directory called "log" and pattern files will be in a sub-directory called "patterns"

Download latest **csl_patterns-X.X.X-bundle.zip** from http://eselivm2v238l.lmera.ericsson.se:8081/nexus/content/repositories/assure-releases/com/ericsson/oss/services/csl_patterns/

Unzip **csl_patterns-X.X.X-bundle.zip** and copy content of patterns dir to S3 patterns dir.

Configuration:

In the directory you installed S3 is a sub-directory called "etc"

This contains four self-documented configuration files:

S3.ini - the main configuration file

S3L.ini - configuration file for low frequency calls

S3F.ini - configuration file for failure calls

S3I.ini - configuration file for incomplete calls

IPFile.dat – the network description file.

Edit IPFile.dat and set your destinations to tell S3 where you want your events to be sent. If needed, you can edit S3.ini to change the defaults values S3 will use.

Initial test to check installation:

edit S3.ini and ensure the key "NumChannels=" has the value of zero.

This mode is primarily for designing and testing patterns.

Run it with the command line:

```
# java –jar S3-<version>.jar -ft -v
```

And monitor the output for any errors. Then check the log files for any warnings. If everything is OK you can proceed to the next step.

Local test :

This mode is primarily for ensuring correct setup and configuration and gets one or more S3 sessions to talk to one or more S3 listeners. Open two (or more) terminal sessions and change to the S3 directory. Check you network configuration file (etc/IPFile.dat) to see which ports are in use. Start an S3 session in listener mode to monitor the ports. the the command

```
# java –jar S3-<version>.jar -lt -p <startport:endport>
```

Then in different terminals, you can start one or more S3 sessions. If there are no problems, the listener will report something like:

```
~/workspace/S3$ java -jar target/S3-2.1.0-SNAPSHOT.jar -lt -p 8880:8890
About to start listener on ports [8880, 8881, 8882, 8883, 8884, 8885, 8886, 8887, 8888, 8889, 8890]
initializing server
Now accepting connections on 127.0.0.1
H:   252 M:    32486 E:    0 Rt:  1412 R1:  2533 kB:000226 kB1:000421
```

Where H: is the number of headers received (one per channel), M: is the number of messages (events) recieved, E: is the number of errors detected, Rt: is the event rate per second (since start), R1: is the event rate per second in the last second, kB: is the kilobytes received per second (since start) and kB1: is the kilobytes received in the last second.

If the results are not as you would expect, then check the log files for warnings or errors and repeat until you are satisfied.

Feature test – with transmission:

This mode is primarily for ensuring correct setup and configuration.

Change the config file entry “NumChannels=” to a value greater than zero to enable TCP transmission. It will then send all the events for the pattern(s) you have chosen to the destination(s) you have selected. Adjust IPFile.dat to describe your network environment. To use virtual IP’s (which don’t work on windows) set the INI file parameter “DistributionType=” to ‘1’ and the program will interrogate the local NIC’s to find which IP’s are available, make sure you get it working with a specified source and destination before attempting to use virtual IP’s.

Use the same command line as before to execute it and monitor the output and the log files for warnings or errors.

Full End 2 End:

Make sure feature test with transmission works first.

Run the command without the –ft option.

```
  # java –jar S3-<version>.jar
```

You can change the volume of traffic generated by editing the INI file and adjusting the parameter “numCallsPerSec=”. Start with a small number (<50) to make sure everything is working and when you achieve stability, you can increase this until you reach your desired traffic levels.

Note:- S3 is network bound and not CPU bound so once you have specified the highest traffic level your network can cope with, increasing the numCallsPerSec further will simply increase the number of events that get dropped. - edit - Fixed in 1.1.10. If generation thread gets too far ahead of the distribution thread, the program will reduce the number of calls per second until the stable maximum is reached. This will only work is the starting level is low enough to not overpower the system and fill available memory before an adjustment can be made.

Command line arguments

Optional parameters you can use include:

-v for verbose output
-i <path> to specify an alternate ini file to use. Default is etc/S3.ini
-t <path> to specify an alternate topology file to use. Default is topology/LTETopologyFile.xml
-s <key> to specify a section of the ini file to use.
-ft for feature test mode
-p <path> to specify the pattern file you want to use - only meaningful in feature test mode.
-d <time> duration of call in seconds - only meaningful in feature test mode
-lt listener test mode.
-p <port range> to specify the range of ports to listen on - only meaningful in listener test mode.
-ip the public address of the server to bind the listener to - only meaningful in listener test mode.

Example:

```
# java –jar ind.jar -ft –v -p PatternFiles11-June-2014/genDTAccessCalls_2_Idle.pat
```

perform a feature test run using the specified pattern

Patterns:

S3 uses patterns to define the events that make up a call. The S3.ini configuration file includes a section called "Paths" which includes a parameter called "PatternDir" which tells S3 which directory to look in for pattern files. S3 will look at each file in that directory and if the ini file contains an entry with the same name, and a frequency greater then zero, it will load it as a pattern to use to generate calls.

Pattern configurations can be maintained in separate ini file as below. S3 will look for the file in the specified "PatternDir"
and load the patterns along with patterns listed in S3.ini file.
For example :
PatternINIFileName="patterns.ini"

For failure patterns: PatternINIFileName="patternsFail.ini"

For incomplete patterns: PatternINIFileName="patterns_incomplete.ini"

Pattern configuration example:

If the directory "Patterns" contained the files, pat1.pat, pat2.pat, pat3.pat, pat4.pat and pat5.pat and the ini file contained a section such as:

```
[Paths]
PatternDir="Patterns"
pat1.pat=3
pat2.pat=50
pat4.pat=49
pat5.pat=-1
```

S3 would load and parse three pattern files. The file pat3.pat would be ignored because it is not in the INI file and the file pat5.pat would be ignored because its frequency is less than one. Any problems encountered parsing the patterns will be reported in the log files. The frequencies are totaled and then the patterns are shuffled so the calls will appear in no particular order. In this case they total 102 which means out of every 102 calls, 3 of them will follow the pattern in pat1.pat, 50 of them will follow pat2.pat and 49 will follow pat4.pat.

Pattern Files:

Call patterns are defined using the following keywords. The character '#' is used to start a comment.

include = <path>
insert the text from the named file at this point in the pattern. This can have an optional repeat count for example
include=X2HO_X2HI.pat, 3
would include the instructions in the named file three times and
include=X2HO_X2HI.pat, r(10)
would include the instructions in the named file a random number between 1 and 10 times.

id = <event_id>
the id of the event to generate. Applies to all subsequent commands until the next id is specified.

offset = xx
the number of milliseconds after the previous event before this one can be sent. This is a hint, not an exact figure.

debug=<some text>
This text will appear in the log file.

period = nn
the last specifed event is periodic and will be repeated every x seconds until the call completes. The first periodic event starts the sequence of periodic events, the first non-periodic ends the sequence of periodic events. (Note:- Period is in seconds, offset is in milliseconds!)

set = <parameter>,<value>
set the specified parameter of this event to the specified value.
<parameter> is a valid settable parameter of the current event as specified in the schema used to generate S3.
Where a parameter exists more then once in an event, you can use an array suffix to specify which entry you want, starting from zero.
<value> can have four formats:
direct implies a number, eg. 123
"" implies a string, eg. "abc"
b"" implies a byte array, eg b"00 2C DA 14" in the format of hex pairs (the spaces are optional).
r(min,max,len) produce a random value x of len bytes such that min <= x < max.

gummei = invalid/unavailable
This is optional keyword that can be used to generate invalid/unavailable gummei values for this pattern.

Pattern Example 1:

CallSetup2ReleaseFail1.pat:

```
# example of a comment
id=INTERNAL_PROC_RRC_CONN_SETUP
id=INTERNAL_PROC_S1_SIG_CONN_SETUP
id=S1_INITIAL_UE_MESSAGE  # end of line comment
id=S1_INITIAL_CONTEXT_SETUP_REQUEST
id=INTERNAL_PROC_INITIAL_CTXT_SETUP
id=INTERNAL_EVENT_MEAS_CONFIG_A3
id=INTERNAL_EVENT_MEAS_CONFIG_A5
period=20  # periodic message to be sent every 20 seconds
id=INTERNAL_EVENT_UE_MOBILITY_EVAL
period=15
id=INTERNAL_PER_RADIO_UE_MEASUREMENT
offset=150 # delayed message to be sent 150 millisecs after previous message
period=20
id=INTERNAL_PER_UE_TRAFFIC_REP  # stuff
offset=250
period=25
id=INTERNAL_PER_UE_RB_TRAFFIC_REP
offset=125
id= INTERNAL_PROC_UE_CTXT_RELEASE
set=EVENT_PARAM_S1_RELEASE_CAUSE,r(1,32,1) # set release code to a 1 byte value between 1 and 32
set = EVENT_ARRAY_ERAB_SETUP_REQ_PCI[4],"abc" # set fourth PCI to "abc"
set=EVENT_ARRAY_ERAB_RELEASE_REQ_QCI[3],b"03 01 C2" # set some bytes
```

Cell Mobility:

New from version 1.2

S3 patterns now support the concept of a session that moves between cells. When S3 generates a call from a pattern that includes an event with the ID "CHANGESOURCE_EVENT", it will get the global cell Id of a neighbor of the current cell (detailed below) and change the key values for subsequent events as if they were generated on this new cell. the following is a minimal example of such a pattern:

```
# set up a call
id=INTERNAL_PROC_RRC_CONN_SETUP
# send a CTUM msg to tell the world where we are
id=CTUM_EVENT
id=S1_INITIAL_CONTEXT_SETUP_REQUEST
id=INTERNAL_PROC_INITIAL_CTXT_SETUP
# Perform an X2 handout to another cell
id=INTERNAL_PROC_HO_PREP_X2_OUT
id=INTERNAL_PROC_HO_EXEC_X2_OUT
# release local resources
id=X2_CONTEXT_FETCH_RESPONSE
id=INTERNAL_PROC_UE_CTXT_RELEASE
# Change to another cell
id=CHANGESOURCE_EVENT
# tell the world we are at the new cell
id=CTUM_EVENT
# receive call from original cell
id = INTERNAL_PROC_HO_PREP_X2_IN
id = INTERNAL_PROC_HO_EXEC_X2_IN
# release the call
id = INTERNAL_PROC_UE_CTXT_RELEASE
```

Selection of cell to move to:

S3 does not (currently) support cell locations so it orders them as they are defined, so the third cell is closer to the second and the fourth then it is the first or fifth. S3 uses the configuration parameter "NumNeighbours" (default 5) to indicate the average size of the jump to make when processing a "CHANGESOURCE_EVENT". Note that "NumNeighbours can be negative to force S3 to always jump in the same direction, useful for emulating transport corridors etc.

Call duration rules setup:

Call duration is only meaningful for patterns that include periodic events. Patterns that do not contain periodic events should be handled by a separate instance of S3 so that they do not distort the proportions of calls of different durations.

S3 use call duration rules to specify the proportions of calls of different lengths that it will generate. These rules are specified in the ini file. Up to 100 rules can be defined using the keys “Duration0=” to “Duration99=” where each rule gives the length of the call and the number of calls of that length.

Call length is 1 second or greater and only the first 1000 calls are considered.

For example, by giving the rules:
Duration1=45,50 # 50 calls of 45 seconds
Duration2=75,20 # 20 calls of 75 seconds
Duration3=5,2 # 2 calls of 5 seconds

S3 is being told that for every seventy two calls generated, fifty will last 45 seconds, twenty will last 75 seconds and two of them will last only five seconds.

If the total number of calls described exceeds 1000 or if low frequency calls account for less than 0.5% of the total then the duration rules to use can be calculated using the spreadsheet Call_Duration_Rules_Setup.xlsx, and configure your S3.ini files as per the suggested duration rules and target CPS rates.

NOTE : At least one call duration rule is required.

NOTE: A warning will be generated in the log file if the longest call specified can not be accommodated by the currently defined Q size. The default Q size has a default of 10,000 entries and is controlled by the "QueueSize" parameter. One entry exists for each "DistMonitor" time slot, so at a default of 125 milliseconds, the default queue can handle calls up to about 1250 seconds long (20 minutes), If DistMonitor is changed to 100 milliseconds, then the longest supported call is about 16 minutes. To support a call of 24 hours with dist monitor at 125 milliseconds would require a QueueSize of over 691,200 (24 * 60 * 60 * (1000/125)) which will have corresponding impacts on memory usage and garbage collection.

Invalid ENBS1APID values:

It is possible to configure S3 to send invalid values for the enbs1apid field. In the ini file being used, set the parameter "corruptCallRatio" to set ratio of calls to be checked, ie one in ten, or one in ten thousand. The default is zero which means no calls will be checked. Also in the ini file being used, set the parameters "corruptCallEvent1" to "corruptCallEvent9" to set the Events to look for in the calls being checked. You can

specified the name or the code. If the event exists in the call being checked and contains an enbs1apid field, it will have its invalid bit set to true. Other events in the same call will not be affected.

For example, by adding the settings below to the in ifile being used, S3 will check every fiftieth call generated and if it contains any of the specified events, they will have the invalid bit of their ENBS1APID set.

```
corruptCallRatio=50
corruptCallEvent1=X2_HANDOVER_REQUEST
corruptCallEvent2=S1_INITIAL_CONTEXT_SETUP_REQUEST
```

Impact: There is a small increase in processing involved which is proportional to the number of messages being checked. For systems already running near to maximum capacity, it might be necessary to check and, if necessary, adjust upper limits if it is planned to set a large portion of the messages to invalid.

Advanced configuration:

Values specified in ini file can be overridden on the command line by using the -D<property=value> option.
If used, this option must appear before the jar file is specified and the property must exactly match a property specified in the ini file.
For example, if the ini file contains the properties numDest=10 and FirstDest=0, then an instance of S3 could be started which only used the third and fourth destination port by specifying the command line as:

```
# java -DnumDest=2 -DFirstDest=2 -jar S3-<version>.jar
```

Topology

The simulator reads a topology file for each eNodeB to get various parameters required to generate events. Topology files are in the 'topology' directory which can overridden with the '-t <path>' command line option. Topology files are named <filePrefix>_MeContext_<enodbname>.xml

To create new topology files, the following command line can be used:

```
# java –jar S3-<version>.jar –wt [options]
```

Options are:
-t <path> topology directory to write to, default "topology"
-f <text> filePrefix to use, default "SubNetwork_ONRM_ROOT_MO_R".
-p <text> prefix for enodeB name, default "ABC_"
-s <text> suffix for enodb names, default "_Town"
-mcc <num> MCC code to use, default '410'
-mnc <num> MNC code to use, default '310'
-h <path> template file to use for EnodeBs, default "template_head.xml"
-b <path> template file to use for Cells, default "template_body.xml"
-c <num> number of cells per eNodeB, default is 5.
-ctype <FDD/TDD/FDD|TDD> type of cell to generate per eNodeB i.e. FDD(EUtranCellFDD),TDD(EUtranCellTDD),FDD|TDD (mixture of EUtranCellFDD and EUtranCellTDD). By default all the topologies will have FDD(EUtranCellFDD) type of cells.
-n <num> number of eNodeBs. If not specified, it will create an entry for each virtualIP on the system.
-1 <num> first eNnodeBId, default 1.

ENodeB names are generated as prefix+enodbId+suffix. File names are generated as topologyDir+'/'+filePrefix+'_MeContext_'+eNodeBName+'.xml'. Cell names are generated as MCC+MNC+'_'+ENODBID+'_'+CELLID.

The topology for eNodeB's and cells uses the template files and replaces the fields marked xxx so it is possible to change the rest of the topology to suit your requirements.

For example, to create the topology for 789 eNodeB's with 4 cells each and the first one starting at number 1234:

```
# java -jar S3-<version>.jar -wt -c 4 -n 789 -1 1234 -t myDir -f SubNet23 -p MAL -s _ville -mcc 353 -mnc 21
```

will create 789 files in the directory 'myDir' named SubNet23_MeContext_MAL001234_ville.xml to SubNet23_MeContext_MAL002023_ville.xml with cells named 35321_1234_1. The values that will be replaced in template files are:

$ENODEBNAME$
$ENODEBID$
$CELLNAME$
$CELLID$
MCC
MNC
$MNCLEN$
$IPV4ADDR$
$IPV6ADDR$

$PHYSCELLIDGRP$
$PHYSCELLID$
$EXPHYSCELLIDGRP$
$EXPHYSCELLID$
$RRCCONNREESTACTIVE$

To set up an emulation of the topology behaviour of ENIQ-M, please look at the page: ENIQ-M emulation with S3

Integrating S3 with CSL-M Topology server , please look at this page, S3 Integration with CSL-M Topology

NOTE : To generate topology files with **FreqBand mfbi** attribute support, use "template_body2.xml" template file for cells.
For example, to generate 100 node topology files with FreqBand mfbi attributes use

```
#  java -jar S3-<version>.jar -wt -n 100 -b template_body2.xml
```

Non-OSS topology

From version 1.1.105 onwards, it is possible to provide a CSV file with basic topology information required for events. This is useful for ENM style environments where topology files are no longer used. To activate this feature, set the parameter **UseTopologyList** in the ini file to the name of file containing the information you want S3 to use. An example of such a file is shown below. The parameters **NumChannels** and **FirstChannel** s till behave as normal so the same topology list can be shared among many instances of S3. The first line of the file that is not a comment must contain the MCC and MNC that S3 should use. This is required to generate the Global Cell Ids used by events. Any errors parsing this file will show in the logs.

```
# comments start with #
# first non comment line set mcc and mnc
# mcc - 3 digit country code
# mnc - 2 or 3 digit network code
319,410
#
# Next lines give eNodeB name, eNodeB Id and number of cells
Node One, 1, 3
# Node is called "Node One", its id is 1 and it has three cells
two, 2, 2
three, 3,3
four, 4, 4
five,5,5
id and cell diff, 6,7
```

Test modes

Three test modes are built into S3 to help set things up smoothly. Feature test mode (described above) and Listener mode and Netty listener mode.

Listener Mode:

Listener mode is a simple listener that can accept events from S3 and report counts and rates. It can be very useful to establish connectivity between two systems.

To start S3 in listener mode, add the command line option '-lt' and use the '-p' to specify the port or range of ports to listen on.

```
 # java -jar S3- <version>.jar -lt -p 8880:8889
```

will start S3 and have it listen on the localhost to ports in the range 8880 to 8889.

The output form listener mode (subject to change) looks like :

```
About to start listener on ports [8880, 8881, 8882, 8883, 8884, 8885, 8886, 8887, 8888, 8889]
initializing server
Now accepting connections on 127.0.0.1
H:  8000 M: 19566174 E:    0 Rt: 13553 R1: 13873 kB:002519 kB1:002577
```

where H is the number of header messages received. M: is the number of events received. E: is the number of errors detected. Rt: is the event per second rate since the program started. R1 is the event per second rate over the last second. kB is the kilobytes received per second since the

program started and kB1 is the kilobytes per second received in the last second.

If the sender is on a different host, the listener will not see it because it is bound (by default) to the localhost. To correct this, you must specify the public address that other hosts use to see this server with the '-ip' option, such as :

```
# java -jar S3- <version>.jar -lt -ip 10.45.193.151 -p 8880:8889
```

The value for the IP parameter is the value used to ping this host from another server. Note:- It is tempting to use the unqualified name (eg. atrcx3615) instead of the IP address, but this is usually mapped to 127.0.0.1 (or ::1) in etc/hosts and so will not allow access for remote servers, even though they can ping using it. The fully qualified name should be ok to use.

An optional parameter '-s1id' can be added which will instruct S3 to show the count of events with an unavailable or invalid enbs1apid. (Defined as the first bit of byte 20 being set in event with a length greater than 40 bytes.)

Netty Listener Mode [File based S3]:

Netty listener is a listener which accepts events from S3 and generate binary files from the events. For every channel a file is created. Each channel relates to an eNodeB and will result in a connection being made to the listener. To enable S3 to send the enode Name through each channel, change the config file(S3.ini) entry “SetEnodeName=” to value "1".
Sending enodeName is mandatory for listener, as that is the unique id between different channels.

To start S3 in Netty listener mode, add the command line option '-nlt' and use the '-p' to specify the port or range of ports to listen on, use '-o' to specify the output location for binary files and '-r' to specify the rop duration for output files.
java -jar S3- <version>.jar -nlt -p 8880:8889 -r 5 -o /tmp
will start S3 and have it listen on the localhost to ports in the range 8880 to 8889 and will generate binary files at /tmp with rop period 5.

Note :

While starting : First start S3 in Netty Listener mode and then start the transmission mode of S3.
While stopping : Terminate S3 with transmission mode and then terminate the netty listener.

Feature Test Mode:

The command line option '-ft' puts S3 into feature test mode. This mode is used to generate a single call following a single pattern. The other command line options available in this mode include:

Options:

-i <path>	the ini file to use
-p <path>	the path to the pattern file to use
-d <num>	the duration of the call to generate. This will also cause events to be sent real time and not as quickly as possible.
-t <path>	the topology file to use
-v	verbose output
-f1 <text>	the ffv/fiv field 1 to use
-f2 <text>	the ffv/fiv field 2 to use
-f3 <text>	the ffv/fiv field 2 to use

Logging:

By default, S3 logs its output to log/S3.log which are rolled daily. The default file used can be overridden for an instance of S3 by specifying an alternative with the -DS3LOGFILE option:

```
# java -DS3LOGFILE=mylog -jar S3-<version>.jar -v
```

will produce output logging in log/mylog.log.

The default is to log at "debug" level, but this can be changed with the by specifying the logging level required. For example:

```
# java -DLOGLEVEL="info" -jar S3-<versions>.jar -v
```

Performance Monitoring and Adjustment:

By default, instrumentation and performance monitoring is run every 10 seconds and logged to a file called log/S3_instrumentation.log. This file is

rolled daily. The default file used can be over ridden for an instance of S3 by specifying an alternative with the -DINSTFILE option:

```
# java -DINSTFILE=myinst -jar S3-<version>.jar -v
```

will produce instrumentation output in log/myinst.log. Performance monitoring and adjustment is required because if S3 generates calls consistently faster then the events can be distributed, the internal buffers will fill until the available memory is exhausted, causing S3 to fail. If you are unsure of the capabilities of your hardware, S3 can be configured to start slow and ramp up to its highest sustainable level by setting the S3.ini parameter 'initCallsPerSec' to a very low figure (such as 25) and the parameter 'numCallsPerSec' to an optimistic figure such as 500 on a laptop, 5000 on a medium server and 15000 on a high speed blade. S3 will then run faster and faster while the events being generated can be distributed. If external factors temporarily reduce the transmission capability, S3 will reduce the rate at which calls are generated in an attempt to compensate. After the interruption has passed, S3 will try to increase its performance again to the maximum it can sustain or to the target rate, which ever is lower.

Scalability:

To simulate a large network, first ensure the appropriate topology is available, generating it if necessary. Trial and error will indicate roughly how many events per second a single instance can support in a given hardware environment, but increasing the number of channels used improves performance. Trying to squeeze everything through one pipe is much harder the feeding many pipes.

Example One: Support an 8000 node system with 5 instances of S3, each instance to have its own log file. This example uses 5 cells per eNodeB to produce a 40000 cell network. The instances can be run on different hardware as long as they access the same configuration and topology files, or some or all of them can be run on the same server.

1. create the topology for the network with the command:

   ```
   # java -jar S3.jar -wt -n 8000
   ```

2. Check S3.ini is appropriately configured and include settings for NumChannels and FirstChannel
3. start the 1st instance of S3 with the command:
   # java -DNumChannels=1600 -DFirstChannel=1 -DS3LOGFILE=S3_1 -jar S3.jar
4. start the 2st instance of S3 with the command:
   # java -DNumChannels=1600 -DFirstChannel=1601 -DS3LOGFILE=S3_2 -jar S3.jar
5. start the 3st instance of S3 with the command:
   # java -DNumChannels=1600 -DFirstChannel=3201 -DS3LOGFILE=S3_3 -jar S3.jar
6. start the 4th instance of S3 with the command:
   # java -DNumChannels=1600 -DFirstChannel=4801 -DS3LOGFILE=S3_4 -jar S3.jar
7. start the 5th instance of S3 with the command:
   # java -DNumChannels=1600 -DFirstChannel=6401 -DS3LOGFILE=S3_5 -jar S3.jar

NB: You must create a topology file for every eNodeB (every Channel) you want to simulate

If there are insufficient topology files or insufficient entries in the topologyList then the number simulated will be reduced

Using Multiple .ini files:

In order to meet specific requirements, such as the ratios of short duration calls to long duration calls, or the ratio of successful calls to failure calls, multiple ini files are used.

Note: When using multiple ini files, we must specify the channels used in order to prevent any two ini files using the same channels. The channels used by each ini file must be distinct.
Also, the number of connections to PMStream must not exceed the number of channels specified.

The following is an example of multiple ini files in use:

java -DNumChannels=20 -DFirstChannel=1 -DS3LOGFILE=S3_1 -jar S3-1.1.58.jar

java -DNumChannels=20 -DFirstChannel=21 -DS3LOGFILE=S3_2 -jar S3-1.1.58.jar -i etc/S3F.ini

java -DNumChannels=20 -DFirstChannel=41 -DS3LOGFILE=S3_3 -jar S3-1.1.58.jar -i etc/S3L.ini

The first 20 channels (1-20) are dedicated to success calls. No ini file is specified as S3.ini is the default ini file used.

The next 20 channels (21-40) are dedicated to failure calls. S3F.ini is specified here.

The next 20 channels (41-60) are dedicated to long duration success calls (S3L.ini).

Specifying the channels like this prevents any two ini files using the same channel.