Plexus SIP Test Harness Architecture & Design

79-3524

Revision History

Rev	Date	Purpose	Originator
Draft	2006-06-15	First Draft	QD Tool Team
1.0	2006-07-19	Version for review	QD Tool Team
1.1	2006-08-18	Update design document to reflect alpha code	QD Tool Team
1.2	2006-11-20	Update design document to reflect 1.1 code	QD Tool Team

Table of Contents

1 Requirements [Nathan Ashelman]

1.1 Purpose

The SIP test harness is a software tool that will help us improve code quality through more extensive testing, without requiring intensive capital investment.

Intended tool users are Plexus SIP developers. The tool will permit unit testing of the SIP code, independent of other software components and Plexus hardware. Test scenarios will be scripted, and an extensive library of test cases will be built up over time and used for regression testing. These test scenarios will include those from feature development as well as bug fixes. Regression tests will evolve to fully exercise both expected and error-handling code paths through the SIP code, with the goal to maximize code coverage. Third-party Linux tools will be leveraged to provide debugging, code coverage measurement and potentially performance profiling and code correctness.

1.2 Requirements

- Users:
  - SIP developers
- Test capabilities:
  - Unit testing
  - Call flows testing
  - Exception testing
  - Configuration changes
  - sipa replication & failover
  - Geo redundancy (sipa part)
  - sipa hot upgrade
  - Automated regression testing
  - SIP, GCC, configuration parameter bounds checking/randomization (vary over and beyond allowed range)
- Platform
  - Linux on x86 (Kernel 2.4.x or above)
  - TCL 8.3 / Expect 5.38
  - Ethereal for packet trace
  - Other 3rd party tools if necessary (gdb, gcov, etc.)
- Supported call flows:
  - SIP messages
  - DNS messages
  - GCC call control primitives

- 3rd Party tools:
  - May be either open source or commercial. Preference for open source
  - Tool should be mature & stable; we do not want to have to debug the 3rd party tool.
  - No instrumentation of the Plexus code is allowed (for example for code correctness), at least at this stage of the project.
  - Capabilities (provided by multiple tools):
    1. Code coverage (required)
    2. debugging, breakpoints (highly desired)
    3. CPU performance profiling (highly desired)
    4. Code correctness (desired)
    5. Memory (leaks, null pointer deref, use of uninit mem, double free, …)
    6. Security (buffer overflows, …)
    7. Multi-threading (race conditions)
- Tool development project deliverables:
  - Test harness tool
  - Integration of 3rd party code coverage and profiling tools
  - Base set of test cases
  - Verify tool usability through several test/fix/test case/retest cycles on real bugs
  - Documentation & User training
- Tool source code:
  - Under same source control as Plexus product code(CVS)
  - Target branch for initial release of tool is 6.2.1
- On-going work:
  - Update test harness code as interfaces to SIP stack evolve
  - Add test cases
  - Tool source code and regression test cases will be under source control
- Evolution:
  - Minimize changes to the test script grammar as the tool evolves to avoid re-writing test cases. Thus the script grammar should be decoupled from SIP code changes as much as possible.
  - Tool should be designed to allow extension or component re-use for future LM and GCC test harness tool

1.3 Roadmap (2006-11-13 updated)

Ver.	FEATURES
Alpha (8/14)	§support basic call scenario, such as section 5.2 script § support only 1 SIPp § support multiple simultaneous users § log collection § test script uses IP address, rather than host name (no DNS) § collect SIPA alarms § provide ~10 test case scripts § provide both state-based and sequential scripts § official user manual § 2-level test suites
Beta-1 (9/16)	§ command history (49492) § review Relay & SIPA code changes § check in all harness code & example scripts § support DNS § Pass/Fail based on SIP message content § Pass/Fail based on GCC message parameter values § support multiple tests simultaneously in one user login § provide ~40 test case scripts § updated user manual
Beta-2 (9/30)	§ gdb integration § support SIPp screen display § support multiple SIPp § support interactive test fully § provide more test case scripts § updated user manual § support following GCC primitives & example scripts: EVTSIPWMSGRSP EVTSIPWMSGREQ EVTSIPWMSGCFM EVTSIPWMSGIND (note: used for SIP messages: options, notify, registration)
1.0 (10/30)	§ SIPA fail-over & redundancy (including GCC primitives: AuditReq/Cfm) § provide fail-over test case scripts § support geographic redundancy primitives & example scripts: GeoAuditReq/Ind, GeoCreReq/Cfm, § gcov integration § upgrade SIPp to v 1.1 § updated user manual
1.1 (11/17)	§ Support SIP-T testing (enhance SIPp or replace with Seagull) § multi-level test suites §SUBSCRIBE/NOTIFY/REFER § Integrate valgrind (code correctness checking) § bug fixes § updated user manual § updated design doc

Other requests	§Port to 3.14 §Integrate sipa.out call flow tool by Jerry Nie (3.x) §Integrate SIPp graphical script generator tool by Bruce Wan § provide simple GUI § ddd integration § support log parsing/analysis § call load (to estimate) § state-based script to simulate GCC, use with SIPp/Spectra

2 Architecture Overview

2.1 Architecture Diagram [Steve Yu]

The following figure illustrates major functional modules in SIP test harness. Also, it shows how the SIPA fits into the test harness. This figure is merely a logical representation of the tool. For details on functional modules, please refer to section 3.2.

Please note that the architecture is based on GCC half-call model.

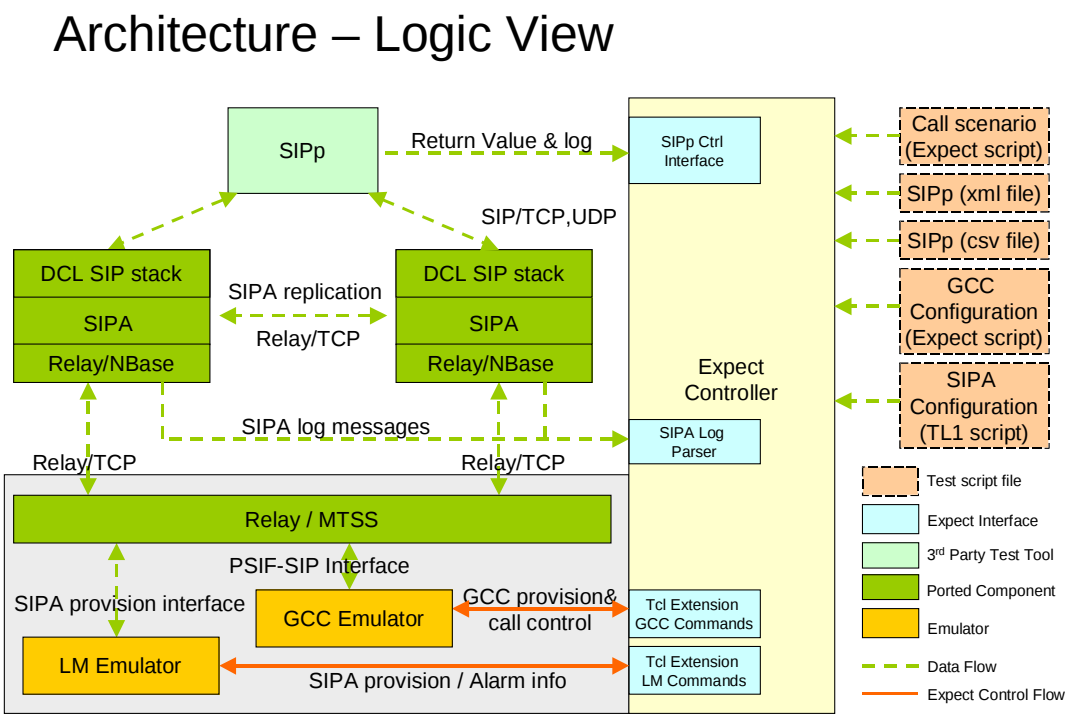


Figure 1: Architecture – Logic View

Process View

The following figure is process view of SIP test harness tool. It illustrates how many processes are involved when test case is running.

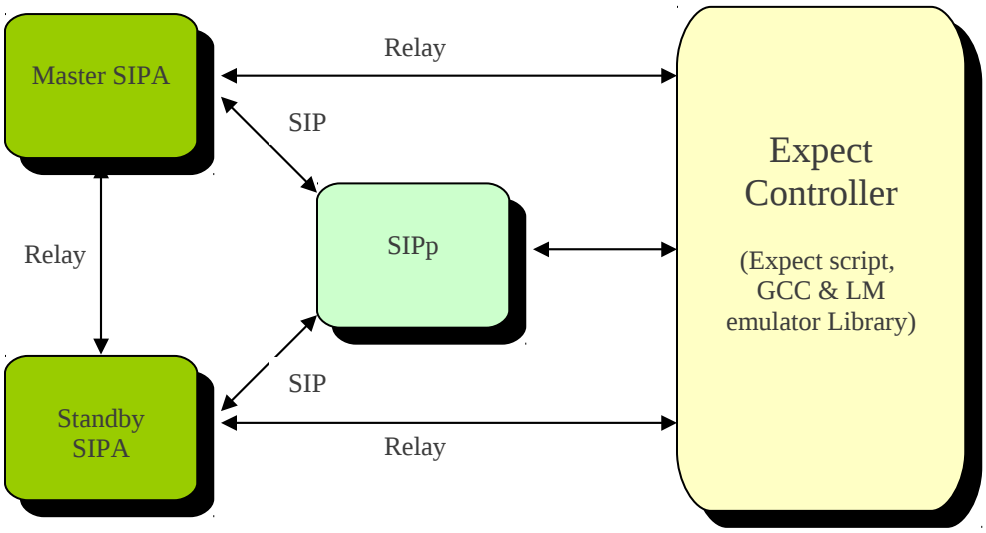


Figure 2: Architecture – Logic View

Please note that harness users are able to define which process needs to be spawn simply by modifying test script. In general, there are four processes running once a test case is launched. They are master SIPA, standby SIPA, SIPp and Expect controller. As for GCC emulator and LM Emulator, neither of them is a separate process. The substitutive way is to compile them into dynamic library, and integrate them into Expect controller through Tcl commands extension.

2.2 Brief Description of Functional Blocks [Steve Yu]

Expect Controller – Expect Controller offers the sole entry to access SIP test harness. The harness user interacts with Expect controller to run test cases. In brief, Expect Controller provides following functionalities.

a. Provide user interface to select test case or test suite scripts, and run them
b. Test scheduler that launches a batch of test cases automatically
c. Test executive that dynamically loads GCC and LM emulator library and executes Expect test script
d. Manage and allocate resources(IP address, port, etc) for simultaneous testing in the same machine
e. Determine whether test case succeeds, based on the criteria specified by testers
f. Save log files of SIPp, SIPA and tool for further manual check
g. Generate detailed test report including test result, statistics data, etc
h. Manage general housekeeping

Once test case or suite is launched, Expect controller spawns master SIPA, standy SIPA and SIPp processes. After the test case or suite is over, Expect controller generates detailed test report according to the following information. The test report includes test result, test statistics data and so on.

a. Return value of SIPp process after SIPp terminates
b. The standard output and log file of SIPp process
c. The log file of SIPA process
d. GCC emulator log and other tool log
e. Code coverage data generated by Gcov

Relay - Relay provides communication channels to exchange information between SIPA and GCC emulator. Relay is able to run over multiple underlying transportation mediums including TCP, shared memory and IPDT. For SIP test harness tool, TCP is adopted with the consideration of keeping consistent with real situation. Similar to real system, Relay on MTSS will be ported and archived into static library for GCC emulator use. And Relay on Nbase will also be ported and archived into static library for SIPA test object.

Tcl Extension Commands(TEC) – Tcl supports both built-in commands and extension commands. Tcl extension commands is a good way to invoke C routines in Tcl environment. So SIP test harness extends Tcl commands to control both GCC emulator and LM emulator. For GCC emulator, the extension commands is call control commands. As for LM emulator, the extension commands are used to provision SIPA configuration.

With the function extension of GCC and LM emulator over time, new Tcl extension commands can be integrated into test harness easily.

Generic Call Control (GCC) Emulator - GCC emulator implements PSIF-SIP interface, through which GCC emulator communicates with SIPA. Similar to real ccs process, GCC emulator sends and receives PSIF-SIP primitives over Relay.

The C-functions that invoke PSIF-SIP primitive would be mapped to Tcl extension commands. Once Tcl extension commands are issued, the corresponding C functions are invoked. So tool users can customize relatively complex call flow using Tcl script. Thus, GCC emulator won't maintain call state since this will be handled by test script.

The parameters in GCC message will be initialized through Tcl script. The GCC emulator parameters configuration involves the parameters of event data structures, such as SipwLiConEvnt, SipwLiCnCfmEvnt, SipwLiCnStEvnt, SipwLiSvcEvnt and SipwLiRelEvnt.

GCC emulator provides Tcl extension commands to set parameters value in PSIF-SIP primitives. Section 5.2.6 gives some description about parameter default values setting.

GCC emulator also help check the validity of the parameters of incoming PSIF-SIP message.

Please note that "GNU compiler collection" is repsented by "gcc", in order to discriminate with "Generic Call Control".

Layer Management (LM) Emulator - LM emulator provides SIPA configuration and alarm collection functions. One-to-one mapping between extension Tcl commands and SIPA provision functions is supported. Tool user provisions SIPA configuration by using TL1 commands. LM emulator also helps collect alarm info from SIPA, and save them to log file. These alarm info will become part of final test report.

SIPA - SIPA is test object. It will be ported from LynxOS to Linux platform. Tool user can get different version of SIPA simply by passing different pre-compile switch.

SIPp - SIPp is a free Open Source test tool / traffic generator for the SIP protocol. It's used to generate SIP message in test harness. It can support complex call flows by using XML scenario files. It features the dynamic display of statistics about running tests (call rate, round trip delay, and message statistics),

periodic CSV statistics dumps, TCP and UDP over multiple sockets or multiplexed with retransmission management and dynamically adjustable call rates. The SIPp homepage link is http://sipp.sourceforge.net/index.html

SIPp is integrated into SIP test harness, with the consideration of never touching source code of SIPp. And SIPp interacts with Expect controller in following way.

a. Expect controller spawns SIPp process and passes configuration files name to it. At least two configuration files feed into SIPp process. One is XML file for SIP message format and call scenario, the other is cvs file for setting variable parameters in SIP message.

b. The child process SIPp informs Expect controller of its termination event. And Expect controller gets SIPp process return value, which indicates whether test case is passed or not from SIPp perspective.

c. To further figure out whether the test case succeeds, Expect controller need to collect and parse SIPp log file.

2.3 Resource Management [Eric Hu]

2.3.1 IP and TCP/UDP ports management

In Plexus switch, the resource management component is used to manage the hardware and software resources. In the SIP test harness tool, we simplify this component because the tool doesn't care about the hardware resources – they are virtual resources in the tool. On the other hand, different from the situation in Plexus switch that only one user can operate the switch at the same time. Harness tool is designed to be able to be used by multiple operators simultaneously. Therefore, we need to pay some attention to TCP/UDP ports allocation and IP address assignment.

Generally, TCP/UDP ports are limited resources sharing with other users and applications. We need to consider carefully how to manage them. The SIP test harness tool supports multiple users on the Linux server. Each user needs at least ten ports to run the harness tool instance: one UDP port for each SIPp instance – maybe more than one SIPp instances are necessary in simulation procedure, two UDP port for SIPA modules for communication with SIPp, two TCP port for SIPA exit usage and five TCP ports for Relay module. The total ports count depends on how many SIPp instances are needed.

If users want to specify the ports by themselves, the harness tool should support them to do so. Port numbers are better to be limited between 10000 and 40000; Otherwise, we would like to let harness tool allocate ports for them automatically. No matter which solution we finally adopt (or support both), there are three choices in realization:

1. Use the Linux server's IP address, supply a mechanism to manage the port number resource. This will cause port conflicts easily for there are a lot of applications running on the Linux server and perhaps some users will occupy ports for their own application usage. For harness tool, it's difficult to estimate which ports are not available.

2. The Linux server can be configured as mullti-homed which can have multiple IP addresses, and we can assign each components one IP address. The conflicts can be reduced dramatically through this solution. But there needs many IP addresses for harness tool usage, and it's also a difficult work to manage so many IP addresses.

3. Configure the Linux server as a muti-homed machine, we only add one more interface(one more IP address), and assign this IP address to SIP test harness, and supply a port management mechanism.

Which solution would be the final plan hasn't been determined yet. The third choice looks better. If harness tool finally adopt this plan, we need to supply a port management mechanism. We will reserve

two port segments from the multi-homed IP address, such as 40000 ~ 50000, 50000~600000 for users who execute the SIP test harness. Because the user id on Linux server is unique, so we will use the user id(500~) as a key value to allocated ports from the first reserved port segment. In case the allocated port number is occupied, we should use "netstat –an" to check whether the port is available or not. If it's occupied by other users, we could allocate the port again from the second reserved port segment.

Note:

1. Each user can only run one instance of the SIP test harness tool
2. The TCP or UDP connection will keep alive even the SIP-A accept failover request

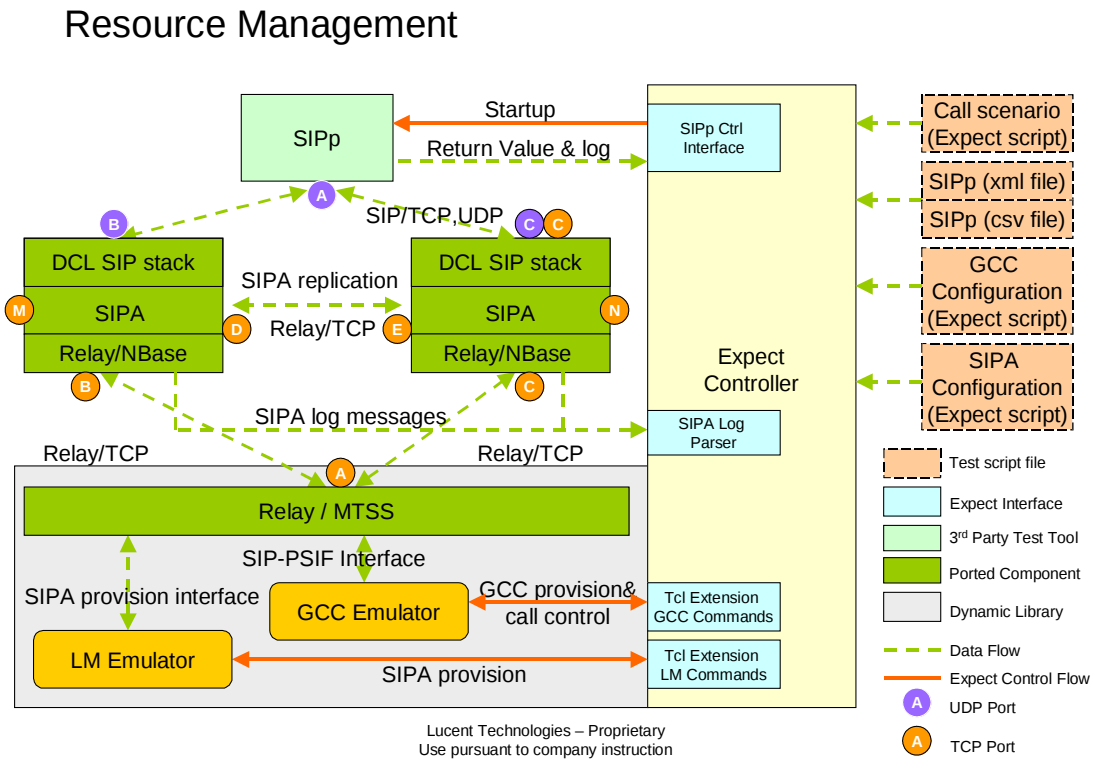


Figure 3: IP port Resource Management

2.3.2 Process management

Harness is a multiple processes program. We need to manage the born and death of these child processes. That's why there is a component named "process management" in harness. For more details, please refer to chapter 3.

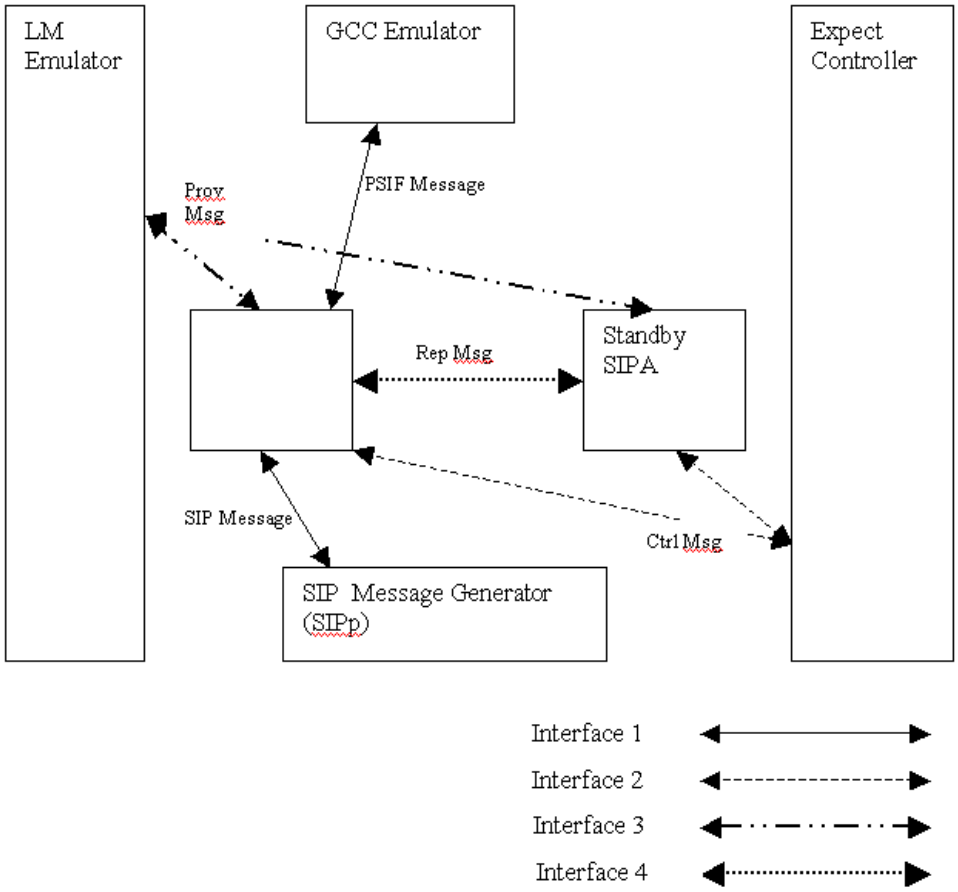
Another topic is the management of Process ID which is used by Relay. Dislike the strict value arrangement in the real switch environment; we only need to guarantee that the Process ID of SIPA distinguishes with the one of GCC emulator. RM will take the responsibility of assigning different ID values for GCC emulator and SIPA.

2.4 Replication & Failover [Raven Bi]

NOTE: Please read other parts of the document firstly before coming into this section. Because we assume the reader has enough knowledge of the test harness architecture here.

The first thing to clarify is that test harness is supposed only to provide an environment to test SIPA replication and failover, but not to construct a full redundant system. That is to say, there will be only one instance for GCC Emulator, LM Emulator and Expect Controller although two SIPA processes will be spawned. What' more, in test harness we have no EMF and GoAhead, so parts of their functionalities will be implemented in expect controller.

Following figure shows the main test harness components from the view of simple redundancy.



LM Emulator

For real switch there are two SLAM instances which will transfer provisioning data to CLAM/GCC/SIPAs on both active and standby cards (standby card first). However, to keep simplicity only one LM Emulator is used in test harness. It will send two copies of provisioning data to both active and standby SIPAs for each TL1 command at the same time. Then there will be two TL1 provisioning responses sent back (one from active SIPA and the other from standby SIPA). LM emulator should check whether the two responses are both OK to determine if a TL1 command has been executed successfully.

Another change to LM Emulator is that if one SIPA is restarted (from Active to Standby, or from Standby to Active), LM Emulator need to re-provision it. On real switch this functionality is taken

cared by LM/DB.

GCC Emulator

There should be no special changes for GCC Emulator to support simple redundancy except that MTSS Relay should be changed to support two relay destinations - after failover it should send PSIF messages (only PSIF messages, provisioning messages should still be sent to both SIPAs) to the new active SIPA by changing destination proc ID. Expect controller (GCC Emulator) should be able to achieve this since it keeps SIPA states information and overall controls the failover procedure.

Expect Controller

In test harness expect controller is in some way like EMF on plexus switch. EMF keeps a complicated state machine to control failover procedure and use GoAhead as a communication mechanism with those processes on active and standby cards. As what we described above, GoAhead is not used in test harness. And expect controller will not maintain a complicated FSM as what EMF does to control failover procedure. However, following information still needs to be kept in expect controller.

- Process ID of two SIPAs.
- SIPA States
- Proc IDs of two SIPAs (used for relay)
- Proc ID of expect controller (used for relay)

The process IDs could be easily gotten while expect controller spawning SIPAs. Proc IDs are currently hard-coded in expect controller since slot IDs in SIPAs are fixed – SP_A_SL (30) and SP_B_SL (31). From the view of expect controller, the slot ID could be used to identify SIPA, which will also be passed to SIPA as a start-up arguments.

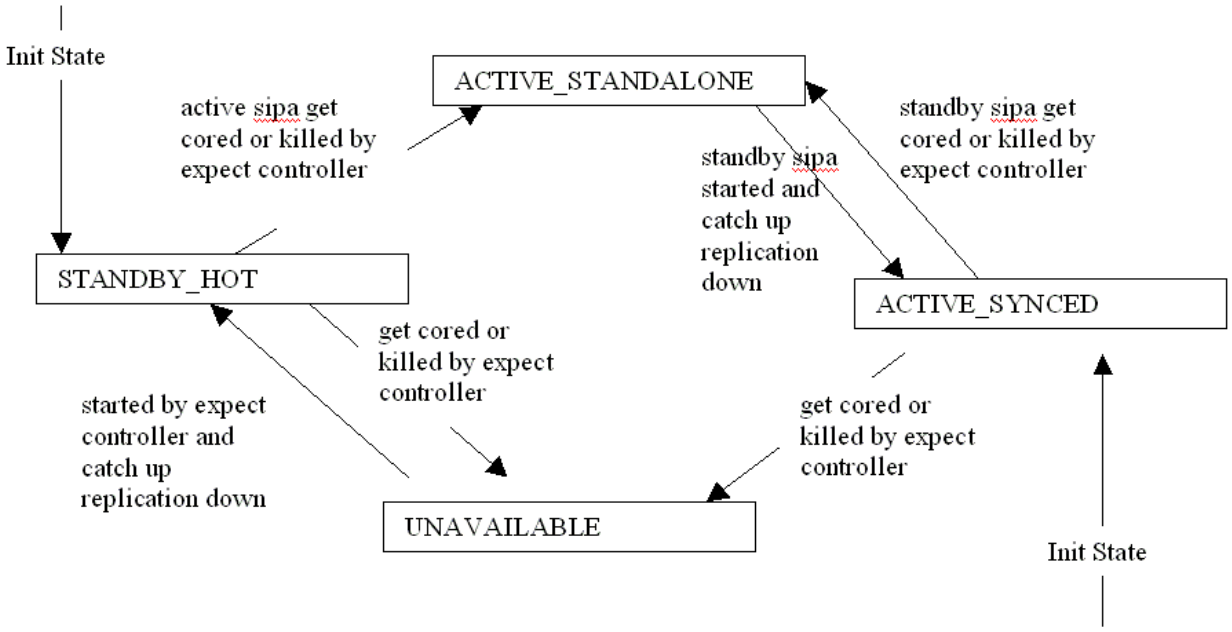
In expect controller, SIPAs have four types of states. They are

ACTIVE_STANDALONE

ACTIVE_SYNCED

STANDBY_HOT

UNAVAILABLE



Above is the FSM that will be maintained in expect controller. Please be noted here,

1. The initial state for SIPAs (after being spawned by expect controller) should be ACTIVE_SYCNED and STANDBY_HOT because in test harness when SIPAs are started for the first time, there will be no catch up replication so we could mark them ACTIVE_SYCNED and STANDBY_HOT directly. However, during the testing (after provisioning) when there is a SIPA restart, the catch up replication will happen and we have to consider this. Current solution is to hard-code a duration (from starting SIPA to marking it as STANDBY_HOT) in expect controller since there will be few provisioning data in test harness and catch up replication will not take too much time.
2. The SIPA states are a little skimp here. It should be enough to test failover scenario but some rainy-day cases are not considered.

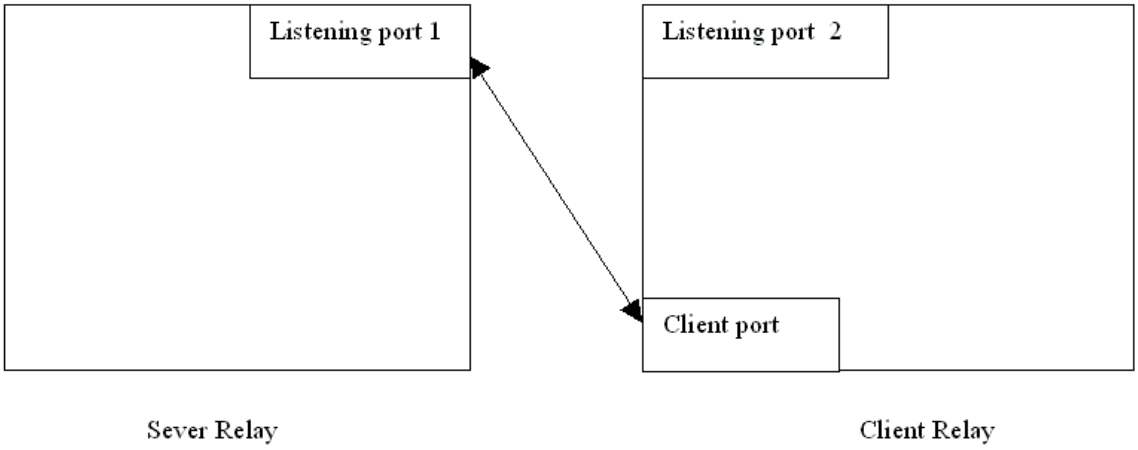
Another functionality that expect controller should implement is the mechanism to get SIPA core dump event and forced switchover event input from user/scripts. The syntax of the switchover command will follow TL1 command SW-TOPROTN-EQPT (SW-TOPROTN-EQPT::SP-{A-B}). After receiving that command expect controller will kill the active SIPA then failover procedure will be triggered. Another option is to let expect controller notify SIPA to set global variable ‘gEmfActive’ to FALSE. The first one is more simple (kill –x is enough) but the second one is more reasonable.

IP Port Resource

For the full description of IP port resource management, please refer to section 3.3.1. Here we just want to discuss the SIPA listening port and NBASE Relay listening port. In real lab environment there are two cards/systems to support simple redundancy. However, in test harness all the processes will run on a single system with one IP address. Following changes are needed to fully use the port resource.

1. For SIPA listening port, don’t need to worry that there will be conflict between two SIPAs. Standby SIPA will not listening on the specified port until it becomes active. So we just let active and standby SIPA listen on the same port.
2. For NBASE Relay listening and client port, we could not use the same one on active

and standby SIPA as what real plexus product does since there is only one IP address in test harness. Please refer to following figure for more details.



The SIPA with slot ID 30 will use server relay. As we could see above, the listening port on client relay side is useless. All the relay related ports will be passed to SIPA as start-up arguments.

sipa <sipa_listen_ip_addr> <sipa_listen_port> <rly_local_port> <rly_remote_port>
<rly_rep_port> <rly_rep_local_listen_port> <rly_rep_peer_listen_port><slot_id>
<active_standby> <log_directory>

For example, for SIPA with slot ID 30, the last three arguments are <Client port> <Listening port 1> <Listening port 2>. While for SIPA with slot ID 31, the last three arguments are <Client port> <Listening port 2> <Listening port 1>.

SIPA

In test harness SIPA should be started using 10 arguments.

sipa <sipa_listen_ip_addr> <sipa_listen_port> <rly_local_port> <rly_remote_port>
<rly_rep_port> <rly_rep_local_listen_port> <rly_rep_peer_listen_port><slot_id>

<sipa_listen_ip_addr> - The IP address of the test machine
<sipa_listen_port> - On which port will sipa listen on (for sip messages)
<rly_local_port> - NBASE relay port used to communicate with Mtss relay
<rly_remote_port> - MTSS relay port
<rly_rep_port> - Client port in last section
<rly_rep_local_listen_port> - Listening port 1or2 in last section
<rly_rep_peer_listen_port> - Listening port 1or2 in last section
<slot_id> - SP_A_SL (30) or SP_B_SL(31)
<active_standby> - Whether this SIPA is active
<log_directory> - The directory where SIPA logs and core will be stored

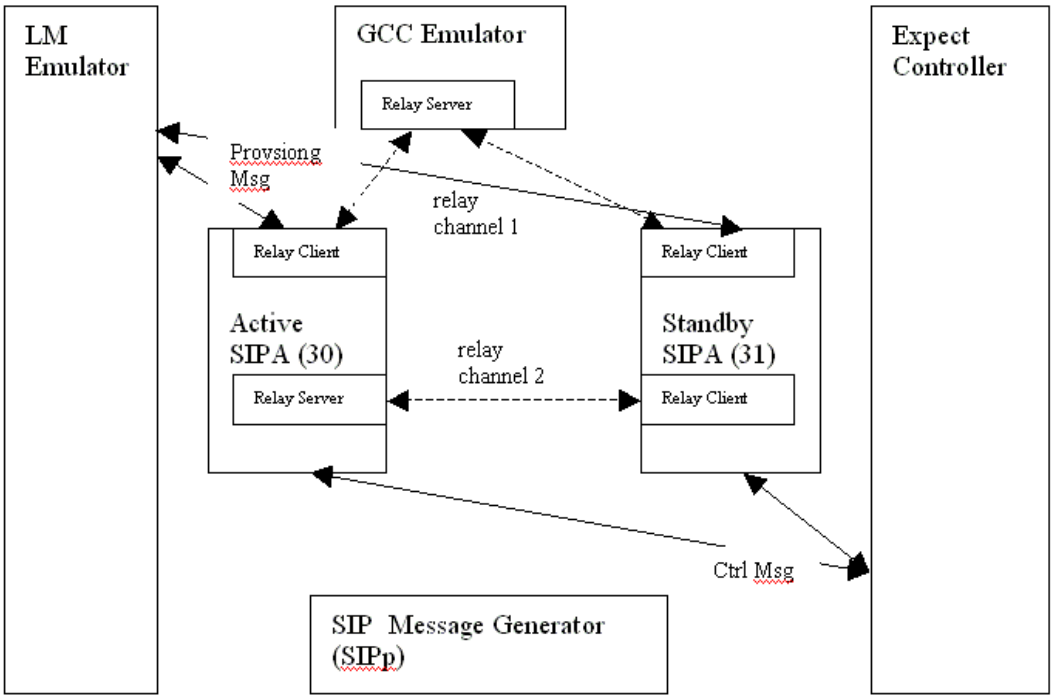
Expect controller will start two SIPAs with fixed slot IDs. And the SIPA with slot ID 30 should be ACTIVE and the SIPA with slot ID 31 will be STANDBY initially. All the other IP address/ports values that will be passed to SIPA are calculated by resource management.

Another change is that we have to hard code dstProcId (the procId of MTSS Relay) in function sipa_yamb_snd_register() although we have different slot IDs for two SIPAs. The reason is that we only have one GCC Emulator.

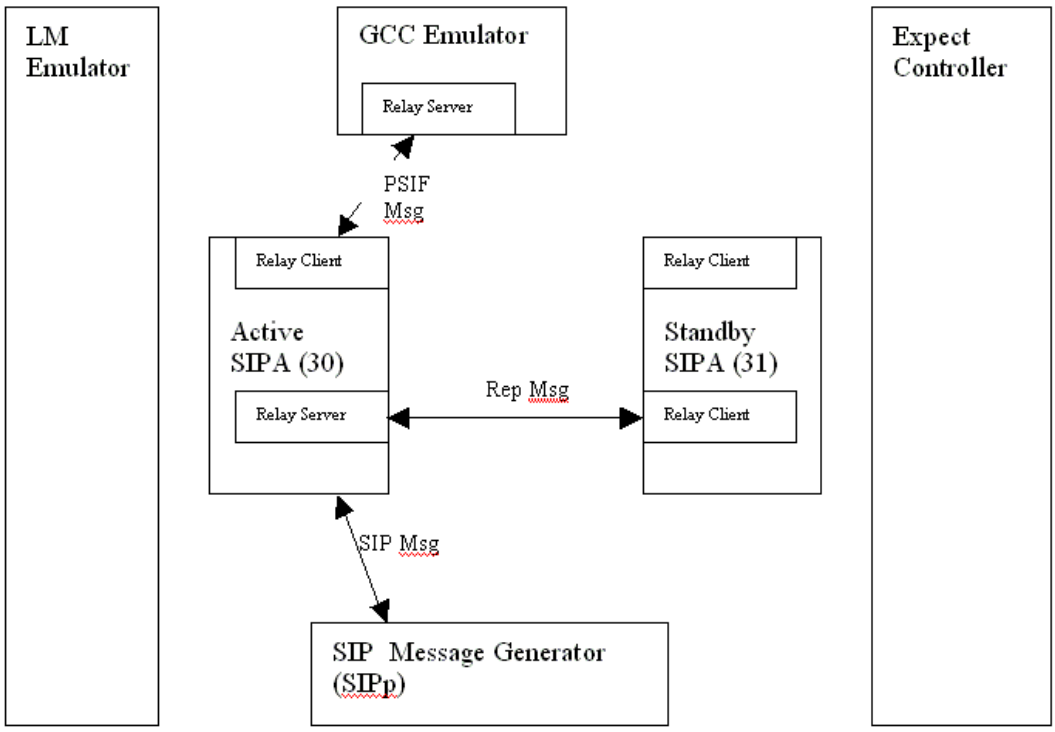
For the replication functionality, there should be no special changes for test harness.

For failover, once Active SIPA gets cored (or killed by expect controller), Standby SIPA SM will get IPS_ATG_RLY_COMM_DOWN_IND in ashm_rcv_app_ips() and call do_goActive() to let itself become the ACTIVE one. DCL provides a good mechanism to do failover and those code are carried to Linux platform with no change. The only exception is that in test harness we have no PSF-SIP so the failover procedure may relies on relay keep alive as described above (see function rly_send_keepalive()).

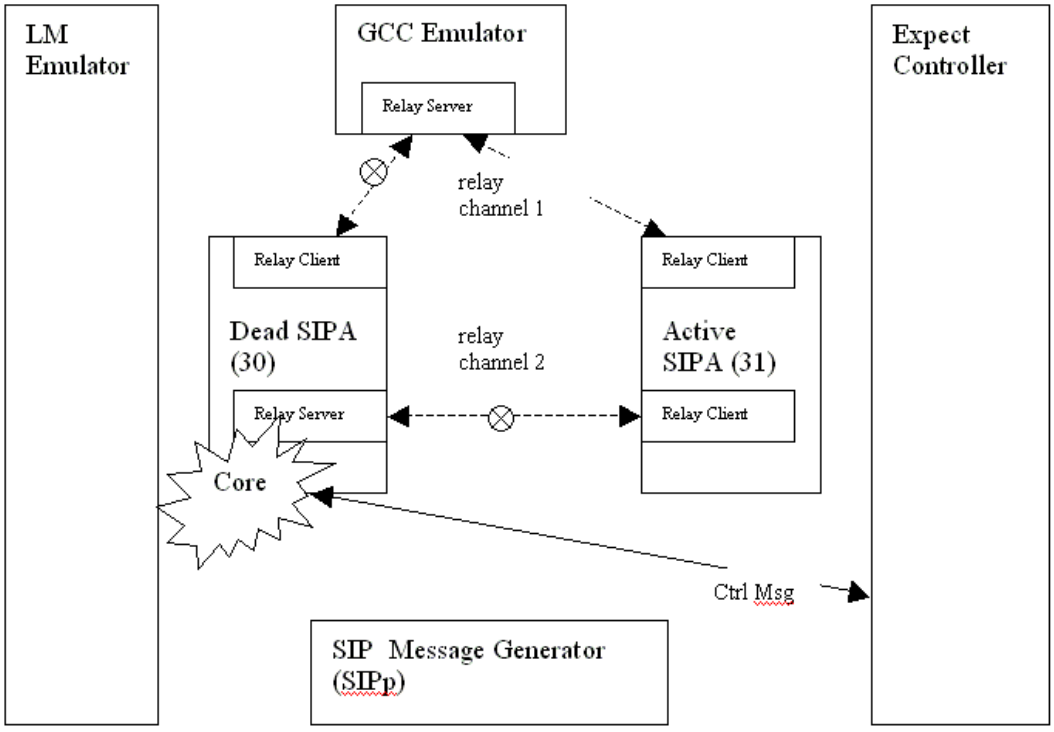
Following four figures demonstrate a typical failover procedure.



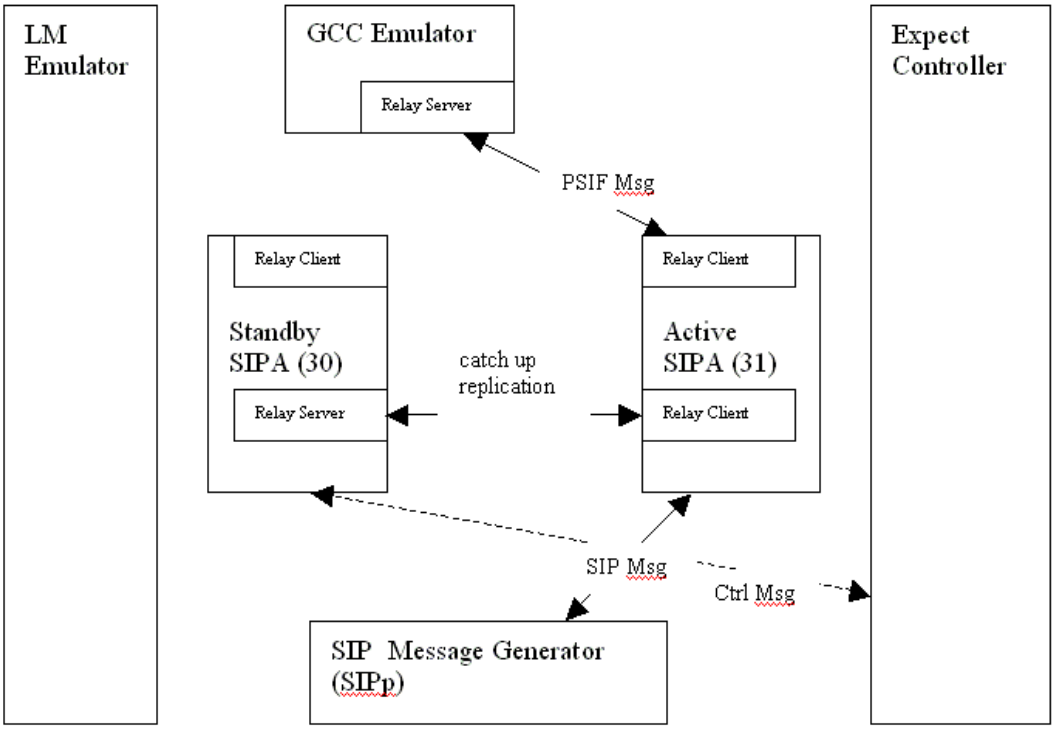
1. Expect controller spawned two SIPAs. One is active with slot ID 30 and the other is standby with slot ID 31. And their states are ACTIVE_SYNCED and STANDBY_HOT.
2. As we could see in above figure, the relay channels between two SIPAs and between SIPA/GCC were established.
3. LM Emulator sent the same provisioned data to both Active and Standby SIPAs.



4. A basic call was made from SIPp to SIPA and got answered.
5. During the processing of the call, related call data were replicated from active SIPA to standby SIPA.



6. Expect controller killed active SIPA (or active SIPA got core dump). The killed SIPA will be restarted by expect controller soon.
7. The SIPA with slot ID 31 got the event that active SIPA was dead so called do_goActive() to become the active one. It will start to listen on specified port to process incoming SIP messages.
8. Expect controller marked the SIPA with slot ID 30 as UNABAILABLE and the SIPA with slot ID 31 as ACTIVE_STANDALONE.



9. Expect controller spawned the SIPA with slot ID 30 again.
10. After 5 seconds (hard-coded value), expect controller marked its state as STANDBY_HOT. The SIPA with slot ID 31 became ACTIVE_SYNCED. SIPA with slot ID 30 is supposed to be able to complete catch up replication with active SIPA in 5 second.
11. SIPp sent BYE to active SIPA and the answered call got released.

2.5 GEO Redundancy Testing Support [Raven Bi]

For simple redundancy, two SIPAs must be spawned with a primary one and a standby one. However, to test geo redundancy, we do NOT need to spawn more SIPAs (for example, 4 instances with 2 as primary MGC SIPAs and 2 as protect MGC SIPAs). The 2 SIPAs (primary and standby) used to support simple redundancy should be enough. Those 2 SIPAs will simulate an active/standby pair on a single MGC. The idea is to divide the geo failover procedure into several phases and test them

separately. It is possible because in geo redundancy protect MGC doesn't keep any dynamic call information, that is to say, there is no call information replication between two MGCs.

From the view of SIPA, only following procedures will be triggered in geo redundancy.

- Supply CRI (Call Recovery Information) data to GCC in ConInd/ConStInd.
We just need to write GCC Emulator scripts to check whether CRI is included in ConInd/ConStInd. In this case primary and standby SIPAs in test harness simulate the SIPAs on primary MGC.

- Process GCC Audit and create GR calls
We only need to write GCC Emulator scripts to send GeoAudReq and GeoCreateReq to SIPAs and verify SIPA action is correct (for example, verify the newly created standing call could be released successfully). In this case the primary and standby SIPAs in test harness simulate the SIPAs on previous protect MGC (new primary MGC).

- SIP Trunkgroup Failover
The OPTIONS 'ping' and session refresh cases could be easily tested with only primary and standby SIPAs.

2.6 Hot Software Upgrade

Hot software upgrade will be supported in future.

2.7 3rd Party Tools Integration

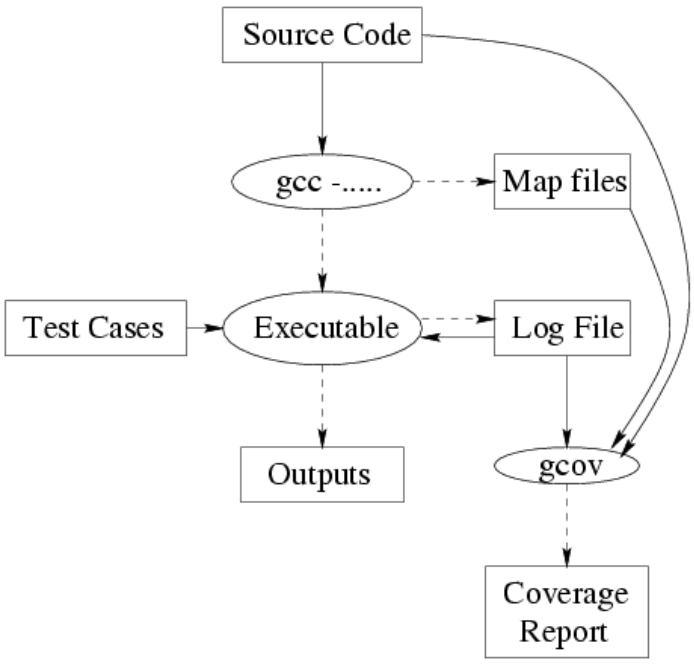
2.7.1 Code coverage [Steve Yu]

Gcov is the GNU code coverage tool, part of gcc, the GNU compiler collection.

Please note that “gcc” means “GNU compiler collection”, and “GCC” represents “Generic Call Control”.

How to use gcov

The following figure illustrates how to use gcov.



ıtial
ign

Detailed procedure is as follows

a. Compile the C program using gcc with special switches (fprofilearcs ftestcoverage)
b. Run the program normally
c. While running, the program will write information to special “log files”
d. After running test cases, run gcov
e. gcov will generate a coverage report listing which lines have been executed

How to integrate gcov into SIP test harness

a. Support the graceful quit of SIPA process
    - SIPA can be terminated gracefully after one test case or test suite is over. It can be configured by Harness user
    - SIPA change: SIPA listens on a specified port, and receives the connection from Expect. Once SIPA receivs quit command from Expect controller, It will quit gracefully
    - Expect controller change: needs to inform SIPA of quit command through socket
b. Gcov requires source code and build binary resides in the same machine. So tool user is required to run test case in SIPA build machine.
c. Summarize code coverage test result
    - Expect controller run Gcov to generate code coverage report for individual file
    - Expect controller summarizes overall result according to individual file result

Table Number 1. Code coverage example

--------Example: Code coverage analysis result (relay test)----------

File `ry_bdy1.c'

Lines executed:34.98% of 526

ry_bdy1.c:creating `ry_bdy1.c.gcov'

File `ry_bdy2.c'

Lines executed:9.02% of 366

ry_bdy2.c:creating `ry_bdy2.c.gcov'

File `ry_bdy3.c'

Lines executed:0.00% of 862

ry_bdy3.c:creating `ry_bdy3.c.gcov'

File `ry_ex_ms.c'

Lines executed:0.00% of 118

ry_ex_ms.c:creating `ry_ex_ms.c.gcov'

…..

2.7.2 gdb Integration [Raven Bi]

In order to debug SIPA effectively, we need to generate debugging information when we compling it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the '-g' option in SIPA makefiles. For example,

```
sipa: $(ALL_SIPT_OBJS) $(LIB_TC_RLY) $(LIB_GENILT)$ (EXEC_LINK_CMD)
      -g -o $(OBJDIR)/$@$(OS_EXEC_EXT)$(ALL_SIPT_OBJS)(OBJDIR)/dchaf.o \
      $(OBJDIR)/dcsm.o $(OBJDIR)/dchm.o $(OBJDIR)/dcnbase.o \
      $(OBJDIR)/dcsck.o $(OBJDIR)/dcnmr.o \
      -L$(TELICA_TOPDIR)/components/ga_dev/lib/lynx \
      -lproc -lTelicaLog -lutil -lhapi $(LIB_SOCKETS) $(LIB_CURSES) \
      $(LIB_NSL) $(LIB_POSIX4) $(LIB_TELICA) $(LIB_PTHREAD) \
      $(LIB_GETLINE) ./siptdbg/librelay.a ./siptdbg/libgenilt.a \
      $(LIB_EXTRA)
```

Using GDB with core files

This is pretty the same as what we do with real product. That is to say, users should start GDB with the core file directly instead of using expect controller. The command syntax is:

gdb program core

Run-time debugging using GDB

To do run-time debugging with GDB, users have to execute '`gdb program`' command in a separate window to start a GDB session with SIPA ***(expect controller may wrap this in a simple command, please refer to section 4 for more details)- Jonathan will update here!!!!!!!!!!!!!!!!!!!!.***

For the detailed list of GDB commands, please refer to GDB manual.

http://www.gnu.org/software/gdb/documentation/

2.7.3 Valgrind Integration

Valgrind is an awarded public software for code correctness purpose. It contains a suite of tools, these tools may benefit us in these fields: One of its advantage is that it's able to detect program's memory leak. Besides that, you could check the cache data, that may be useful under some circumstance. It also can detect the race condition.

The latest valgrind's version is 3.2.1, more useful tools are under develop. For specific, please see:

http://valgrind.org/

2.8 Code Location & Makefiles [Raven Bi]

TestHarnessUpdate script:

A new script named *`TestHarnessUpdate`* was added which is used to check out test harness only. Basically it will check out test harness code, test scripts, SIPA, Relay, logging library and some header files that are necessary to build test harness. The widely used *`TelicaUpdate`* script is not changed so even if users have run *`TelicaUpdate`* before they still need to execute *`TestHarnessUpdate`* to get test harness.

Tool Source Code Location:

For Logging Library, SIPA and Relay (on MTSS and NBASE), code location will keep unchanged since we plan to add condition compiling flags to original code. For Linux porting related code, '`LINUX`' was chose. While for tool related code, '`TEST_HARNESS`' will be used. Please be noted here, in R6.3 (or later) we are porting the whole product to MontaVista Linux and OS abstraction APIs may be provided. If so we will erase '`LINUX`' flag and use those OS abstraction APIs.

Test scripts will be in `TelicaRoot/components/test_harness/scripts/sip`.

For other components, in `TelicaRoot/components/test_harness`

```
test_harness                        - common makefile
test_harness/bin                    - executable files
test_harness/sipp                   - SIPp source code
test_harness/lmgcc                  - lm and gcc emulator
test_harness/expect_controller      - expect controller
test_harness/3rdparty               - all 3rd party tools
```

Makefiles:

We have an overall makefile in directlry 'TelicaRoot/components/test_harness' that could be used to make all test harness executables. Basically it will call following sub-makefiles to make different test harness components.

TelicaRoot/components/telica_common/Simulator/log.mk	- logging library
TelicaRoot/components/signaling/sip2.3/jobs/gnu/*.mak	- SIPA make files. Test Harness specific parts are wrapped with #ifdef SWTH.
TelicaRoot/components/signaling/mtss/Simulator/Makefile	- MTSS make file
TelicaRoot/components/signaling/relay/Simulator/Makefile	- Relay make file
TelicaRoot/components/test_harness/expect_controller/Makefile	- Expect controller make file
TelicaRoot/components/test_harness/lmgcc/Makefile	- lm_gcc emulator make file

2.9 Open Issues

2.10 Close Issues

Issue 1: Does GCC use full call model or half call model?

Use half call model

Issue 2: Does we need to provide GUI interface at present?

Provide GUI interface in future release

Issue 3: Does we use state-based script, sequential script or object-oriented script?

Support both state-based and sequential script

Issue 4: How to get the return value of SIPp? Now we can "spawn" shall, then startup sipp using "send sipp …..\r". The shell will help to get the SIPp return value. If we "spawn" sipp directly, I havn't find out how to get the return value. "wait" looks like meeting the requirement. It normally returns a list of

four integers. The first integer is the pid of the process that was waited upon. The second integer is the corresponding spawn id. The third integer is -1 if an operating system error occurred, or 0 otherwise. If the third integer was 0, the fourth integer is the status returned by the spawned process. If the third integer was -1, the fourth integer is the value of errno set by the operating system.

SIPA, SIPp is spawned by using C program, other than Expect Script.

Issue 5: How to real-time monitor log file?

Tail command should be ok.

3 Functional & Interface Description

3.1 Expect Controller

3.1.1 Expect Controller Infrastructure [Steve Yu]

Expect Controller Infrastructure features following functions.

1. support both interactive test and automated test
   - launch/stop test case or suite by Tcl commands
     i. specific thread for test case or suite execution
     ii. provide “wait for condition” & “time out” mechanism
   - Interactive command is available even when automated test is running
     i. retrieve test status, change SIPA debug level, etc by Tcl commands
2. support state-based test script & sequential test script
   - both of these two kinds of scripts follow Tcl grammar
3. support hard-timing test model
   - timer handler infrastructure, schedule action whenever
4. support event-driven test model
   - event register, de-register, trigger, handler, etc
5. extensibility
   - extend Tcl commands simply by loading so file (i.e. GCC&LM dynamic library)
     i. embed Tcl interpreter into C program
     ii. load user dynamic lib by C program, not Tcl

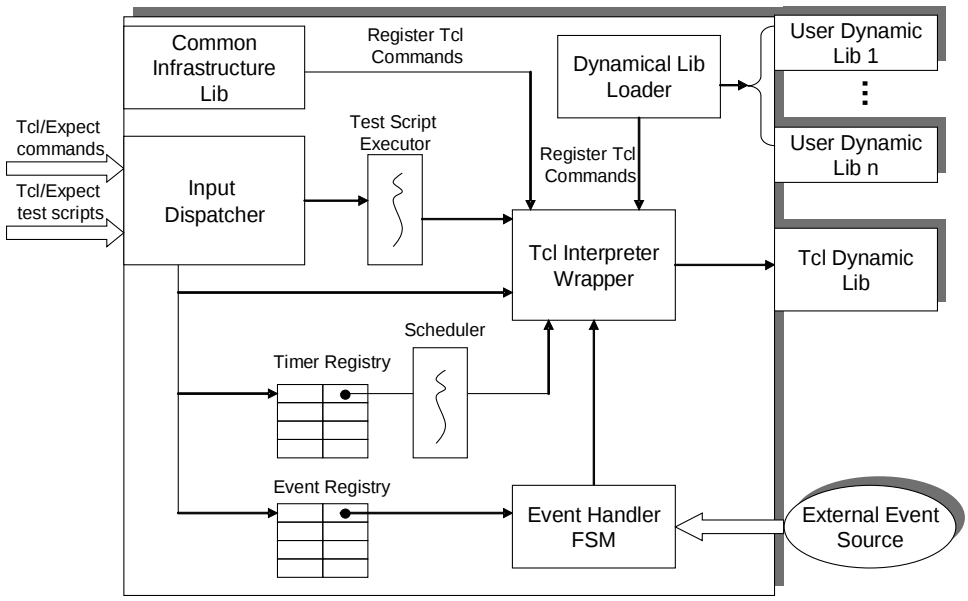


Figure 5: Expect Controller Components

The above figure illustrates main components of Expect Controller Infrastructure.

Dynamic Lib Loader – Dynamic Lib Loader provides the functions to load the dynamic library, and register the extension commands to Tcl interpreter. For SIP test harness, GCC&LM dynamic library is loaded by it during harness initialization. Also, harness user can develop their own specific commands and integrate them into the harness simply by loading dynamic library.

Input Dispatcher – Input Dispatcher provides two functions.

1) accepts Tcl/Expect commands and figure out whether the commands is completed. If the command is completed, pass it to Tcl Interpreter Wrapper
2) read the test script files line by one, and pass the completed Tcl/Expect command to Tcl Interpreter Wrapper

Tcl Interpreter Wrapper – Tcl Interpreter Wrapper is used to wrap internal mechanism of invoking Tcl interpreter, with the purpose of minimizing the complexity of using Tcl interpreter. Also, it's responsible for coordinating using Tcl interpreter to avoid race condition that caused by multithread access.

Event Registry – Event Registry keeps the records of mapping between event type and event handler. Once event comes in, the expect controller would look up this registry, and find out the corresponding event handler, and then fire it.

Event Handler FSM – Event Handler FSM is provided by SIP harness user. Harness user can customize this FSM in Tcl language, and register this FSM to "Event Registry" with the particular event type.

Test Script Executor – Test Script Executor is a specific thread, which runs the test case scripts or test suite scripts. Also Test Script Executor ensures that there is one test task running in test harness.

Timer Registry – Timer Registry. Harness user can register timer, and schedule particular action that needs to be run in future. Test harness is able to support hard-timing test mode by introducing this mechanism.

Scheduler – Scheduler serves as the infrastructure for hard-timing test model. It is a specific thread, which keeps eyes on real system time and Timer Registry. If any timer is expired, the corresponding action would be performed.

Common Infrastructure Lib – Common Infrastructure Lib is a collection of common functions. These functions are registered to Tcl interpreter. So all of them are mapped into Tcl Extension Commands, in order to support more powerful functions. With that, Expect controller has good extensibility since it can be enriched very easily over time.

3.1.2 Resource Management [Jonathan Li]

According to discussion in 3.3, we finally adopt the third solution to solve the port conflict problem. Generally, harness tool will own a unique IP for all users' tool instances. So other applications in the system would not interfere the port allocation of harness tool. The harness tool also doesn't need to check whether the port is occupied by foreign applications.

For the second place, harness tool implements the user id to decide in which range the port numbers are available. The computing method is like this:

user_id * 20 <= TCP/UDP port number < (user_id + 1) * 20

For instance, if the user id is 500, the upper limit of port number would be 10020 and the lower limit would be 10000. That is, 20 ports are available for this harness tool instance. Linux may support more than 60000 TCP/UDP ports. Considering that user_id always starts with 500 and increases 1 for each new user, it allows totally 2500 users to apply the harness tool simultaneously. The total number of actual used ports depends on the call scenario details, especially how many SIPp instances are needed to simulate the call.

For another thing, RM doesn't take the responsibility of assigning port to specific component. Its duty is to provide an available port pool. For specific port-component mapping (etc. which port does the SIPp use), please refer to component design.

As interface design, we provide a port pool allocation method like this:

```
#define PORT_POOL_NUM 20
void allocatePortPool(int iUserId, int iPortPool[PORT_POOL_NUM])
{
        int i = 0, j = 0;
        for (i=0; i<PORT_POOL_NUM; i++)
        {
                iPortPool[i] = iUserId * PORT_POOL_NUM + i;
        }
}
```

Table Number 2. Ports allocation

3.1.3 Process Management [Jonathan Li]

Harness is a program of multiple processes and multiple threads. Both SIPA and SIPp are child processes. Standard Unix API “fork” and “pthread_create” are used for these two purpose.

The procedure of creating a new process is encapsulated in function “spawnproc”. Besides forking a child process and executing a new application (SIPA or SIPp), this function also do many other works like redirect the I/O, initiate the related process node in process list and notify the telnet server to prepare for providing control and display user interface in specific TCP port.

Under some circumstances, SIPA or SIPp may shut down by users or operation system for various kinds of reasons. Therefore, expect controller needs to maintain a state list to manage the status of all child processes.

The implementation of this state list is realized in procutils.c. When harness generates a child process through the method “spawnproc”, a new node will be inserted to this process list. Under two conditions this node may be freed or marked as “exited”: one is through the function waitproc, that means harness is about to detect whether the process is terminated and kill it if it’s present. In the execution of this function, many other works are needed to be done after the process is terminated, releasing the related information in process list is one among them; the other happens when the process is shutdown by some application or operation system other than harness. The operation system will send SIG_CHLD signal to harness, and harness could capture this signal and begin to do corresponding works.

The implementation of Process Management is mainly in files procutils.c and comlib.c.

3.1.4 SWIG [Alex Cao, Caleb Chen]

There are a lot of parameters we should configure in messages transmitted between GCC emulator to SIPA, some parameters are in form of structure and nested structure, and some structures include pointer, union. SWIG is a tool that is capable of wrapping all of ANSI C features including structure, union and pointers. It also supports for almost all of ANSI C++ features. With the help of SWIG, we can combine these parameters into a real structure or other proper types in Tcl, and send them to GCC emulator as an intact one instead of sending field by field. So that is why SWIG comes in. And it also benefits the interface building between Expect and other components. SWIG homepage link is http://www.swig.org/

In test harness, SWIG is used to map C functions and structures in GCC emulator and LM emulator to Tcl extension commands (TEC). In order to do that, SWIG needs an interface file (or C header file), which includes the structure definition and function prototype. After compiling the interface file, SWIG generates a wrapper file. When compiling the GCC emulator or LM emulator, the wrapper file is compiled together. The final result is a shared library, which can be loaded by Tcl directly. Then the structures and functions become the TEC. They can be used to define structures or do what the C functions do.

Here is an example. Structure SipwLiConEvnt (its definition is in Section 4.10) and function sendConEvnt (its definition is in Section 4.7.2) belong to the interface between Expert controller and GCC emulator. Their corresponding TEC are also SipwLiConEvnt and sendConEvnt. An variable of type SipwLiConEvnt can be defined like this:

tclsh>set sipwConEvnt [SipwLiConEvnt]

Command sendConEvnt also has two parameters as its counterpart in C. Its first parameter has type SipwLiConEvnt. So it can be used like this:

tclsh>sendConEvnt $sipwConEvnt $EVTSIPWCONREQ

The full command list and detailed usage are covered in Section 5.1.1.

The problem is which file should be used as the input of SWIG, header file or interface file.

① Using header file is more convenient, but the wrapper file need to be modified. Another problem is that SWIG can't access the variables or functions that defined in the source file but not declared in the header file.

② Using interface file is the way recommended by SWIG documentation. And this way can overcome the shortcomings of using header file. But need to write it manually and maintain it in the future.

③ Using interface file to include original header files.

The third way is preferred. The reasons are:

① We can't require users to modify the wrapper file.

② It needs less maintenance work than the second way.

3.1.5 User Interface

The tool should support interactive mode, in which the tool provides a kind of expect shell as the command line interface to the user.

```
bash> start_test_harness
```

…

after the tool is brought up, an expect shell is ready to the user.

```
test-harness>
```

3.1.6 Graphic User Interface

Will be supported in future

3.2 Relay [Steve Yu]

Relay provides three logical communication channels:

a. Call control channel

Call control channel is used to exchange PSIF-SIP primitive between SIPA and GCC emulator. The channel bridges between SIPA and GCC emulator. SIPA uses relay on Nbase. GCC emulator uses relay on MTSS.

b. Provision channel

Provision channel is used to provision configuration data to SIPA process by LM emulator.

Relay on Nbase is used for SIPA. As for LM emulator side, relay on MTSS is used. In current design, LM emulator shared the same physical communication channel with GCC emulator.

Also, provision channel is used to collect alarm info from SIPA by LM emulator.

c. Replication channel

Replication channel is established between master SIPA and standby SIPA. Both of them use relay on Nbase.

Relay porting

Relay will be ported from real Plexus code. The porting can be split into two parts.

a. Relay on MTSS: needed by GCC emulator and LM emulator

b. Relay on Nbase: needed by SIPA

Note: In some cases, harness user needs to run SIPA step-by-step in GDB. Since under this situation Nbase Relay couldn't send heart beat message to GCC&LM Relay on time. So the keep alive receiving timer of GCC&LM Relay needs to be disabled. Otherwise, the Relay connection would be teared down due to lost of heart beat.

Code Change Strategy

a. To avoid impact on plexus product, compile switch will be used

   I). The tool-specific code change is enclosed by "#ifdef SIMULATOR_TOOL" statement

   II). The OS-specific code change is enclosed by "#ifdef LINUX_PORT " statement

b. MTSS will be compiled into "libmtss.a" static library

c. Relay on MTSS will be compiled into "libry.a" static library

d. Relay on NBase will be compiled into "librelay.a" static library

Touched files list for Relay on MTSS

1. components/signaling/mtss/mt_ss.h
2. components/signaling/mtss/mt_ss.c
3. components/signaling/mtss/ss_acc.c
4. components/signaling/mtss/ss_mem.c
5. components/signaling/mtss/ss_msg.c
6. components/signaling/mtss/ss_task.c
7. components/signaling/sigcom/cm_bdy5.c
8. components/signaling/sigcom/cm_inet.x
9. components/signaling/sigcom/cm_mem.c
10. components/signaling/relay/ry.h
11. components/signaling/relay/ry_bdy1.c
12. components/signaling/relay/ry_bdy2.c
13. components/signaling/relay/ry_bdy3.c
14. components/signaling/relay/ry_tcp.c
15. components/signaling/relay/ry_ptli.c

16. components/signaling/relay/ry_ex_ms.c
17. components/signaling/relay/ry_ptmi.c
18. components/signaling/relay/ry_lsb.c
19. components/signaling/relay/ry_acc.c
20. components/signaling/relay/ry.x
21. components/signaling/relay/smrybdy1.c
22. components/signaling/relay/sigcom/cm_inet.c
23. components/signaling/relay/sigcom/cm_inet.x

How does GCC&LM emulator use Relay on MTSS?

a. Including library of Relay on MTSS by adding "-lry" switch

b. Include header files list: refer to components\signaling\relay\ ry_acc.c

c. Initialize Relay communication channels

- Call SsetProcId function to set process id. SsetProcId prototype:

```
PUBLIC Void SSetProcId(procId)
```

- Call RyCfgChanTCP function to configure TCP relay channel. RyCfgChan prototype:

```
PUBLIC Void ryCfgChanTCP
(
U8 chanId,
U16 chanType,
U32 localPortNo,
U32 remPortNo,
U8 *remHost,
ProcId dstProcIdLow,
ProcId dstProcIdHi,
ProcId rProcId
)
```

d. Send/Receive data over relay

- Call SPstTsk function to send data. SPstTsk prototype:

```
PUBLIC S16 SPstTsk( Pst *pst,  Buffer *mBuf);
```

- Call SRegTTsk function to initialize/register receiving message handler function. Once messages are arrived, the handler function will be invoked. SRegTTsk prototype:

```
PUBLIC S16 SRegTTsk
(
Ent ent,                /* entity */
Inst inst,              /* instance */
Ttype type,             /* task type */
Prior prior,            /* task priority */
PAIFS16 initTsk,        /* initialization function */
ActvTsk actvTsk         /* activation function */
```

)

3.3 GCC and LM Emulator [Katie Zhang, Joey Zhang]

3.3.1 GCC Emulator

Below figure describes the role of GCC Emulator in SIP Test Harness.

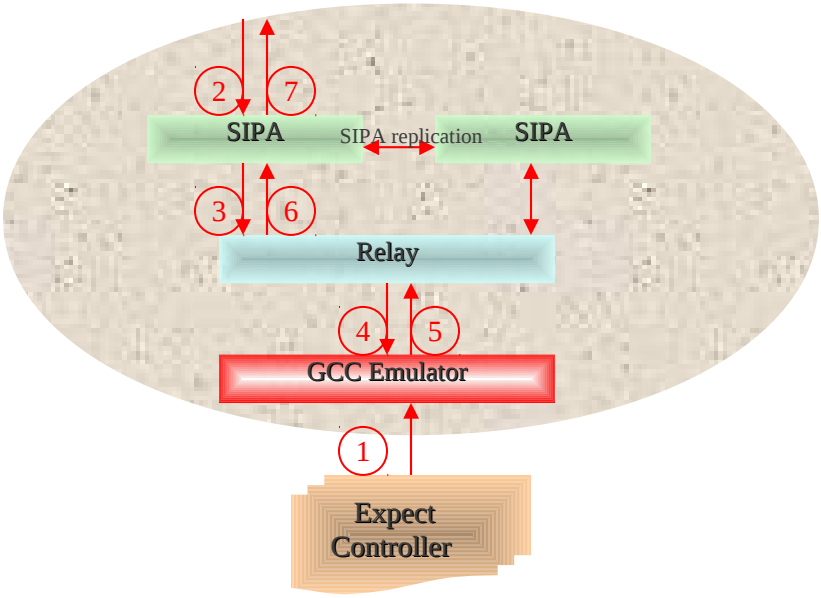


Figure 6: GCC Emulator Data Flow

①Expect controller loads and parses the call scenario file, then call GCC emulator to execute it

②Active SIPA gets messages from SIPp

③SIPA sends the message to Relay

④Relay re-transmits the messages to GCC Emulator

⑤GCC Emulator valid checks the messages and generates the response/request according to call scenario file, then sends them to Relay

⑥Relay re-transmits the messages from GCC Emulator to SIPA

⑦SIPA re-transmits the messages to SIPp

From Figure 6, we can divide GCC Emulator into 2 function modules: GCC Emulator Controller and GCC Emulator Communicator.

1) GCC Emulator Controller

It controls the call logic according to call scenario/configuration file. Its functions are as following:

a. It reads the call scenario/configuration files and parses the information from scenario file to SIPA readable data.

b. It waits SIPA messages and sends SIPA corresponding response/request according to scenario file. It will fill in the default values for messages parameters if they are not

specified in call scenario/configuration file.

c. It sends the response/request to SIPA through GCC Emulator Communicator according to scenario file.

d. It also takes charge of validation check of messages from SIPA and call scenario file, including message parameter values etc.

This module is implemented as part of expect controller.

2) GCC Emulator Communicator

It is responsible for communicating with Relay. It only package/un-packages messages and calls Relay functions to send/receive messages. Its functions as following

a. When it receives messages from Relay, it will un-package the messages and accommodate the messages to GCC Emulator Controller.

b. When it receives messages from GCC Emulator Controller, it will package the messages and calls the Relay functions to send messages.

c. It will only check messages format when the messages arrives.

It will be implemented by C language and compiled to .so. Tcl/Expect will load it through dynamic link library. Its codes are real Plexus codes.

Below figure is the GCC Emulator function modules.

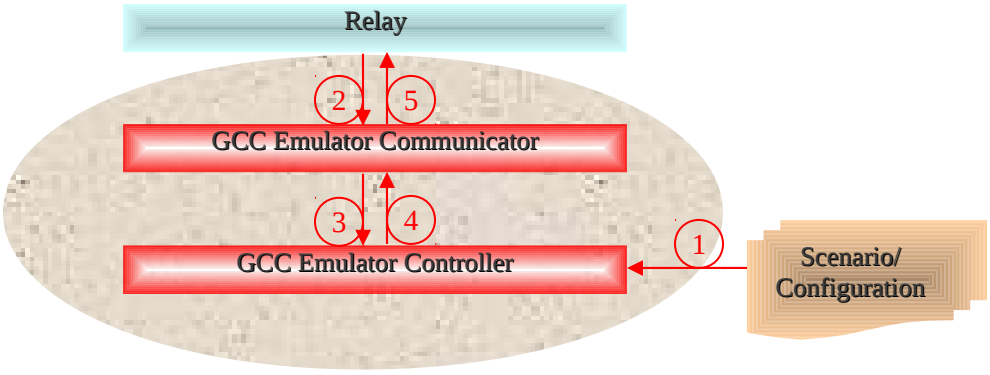


Figure 7: GCC Emulator Function Modules

①GCC Emulator Controller starts up with loading scenario/configuration files

②GCC Emulator Controller calls GCC Emulator Communicator to wait messages from Relay.

③GCC Emulator Communicator receives a message and notifies the GCC Emulator Controller

④GCC Emulator Controller decides what information will be sent out according to scenario file and calls GCC Emulator Communicator to send the message

⑤GCC Emulator Communicator sends the message to Relay

3.3.2 LM Emulator

LM emulator has two main functions. First it parses the SIPA configuration file (text format) when starts up, and then send SIPA readable data structures to SIPA. Secondly, it collects alarm info from SIPA and writes them to log for test report, handles TL1 commands response from SIPA. The configuration data sent to SIPA includes both SIP call configuration data and SIPA debug commands.

LM emulator would do some simple validation for the parameters' value and check the format in TL1 command through TL1 parser.

The LM emulator process will communicate with SIPA using relay.

From above description, we can divide it into 2 function modules: LM Emulator Controller and LM Emulator Communicator.

1) LM Emulator Controller

It is responsible for reading the SIPA configuration file, parses the information from configuration file to SIPA readable data and sends the configuration data to SIPA through LM Emulator Communicator, handles TL1 commands response, accepts SIPA alarm messages through LM Emulator Communicator and writes them to log file.

This module will be implemented in tcl/Expect.

2) LM Emulator Communicator

It is responsible for communicating with SIPA. First it will send configuration data to SIPA. On the other hand, it handles the TL1 commands response from SIPA, receives alarm messages from SIPA, and writes the messages to log file.

It will be implemented by C language and compiled to .so. Tcl/Expect will load it through dynamic link library.

Figure 8 is the LM Emulator function modules.

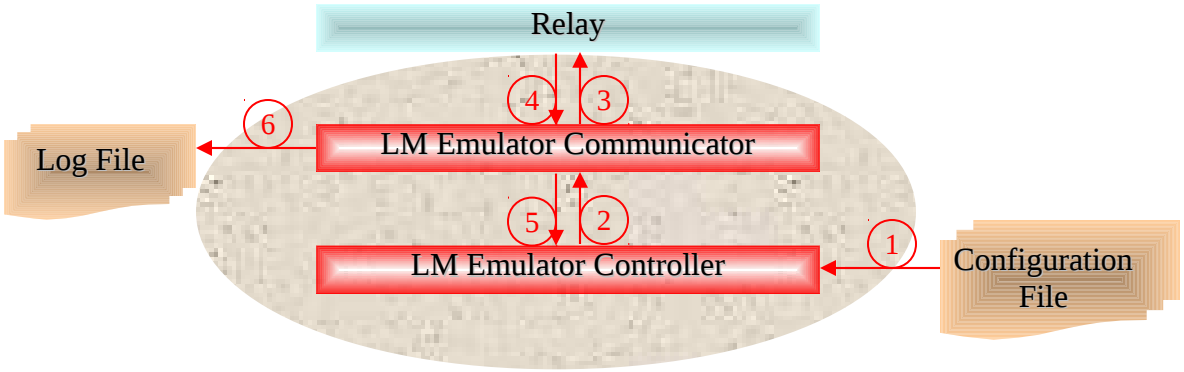


Figure 8: LM Emulator

①LM Emulator Controller starts up with loading configuration files

② LM Emulator Communicator calls LM Emulator Communicator to send provision data

③ LM Emulator Communicator sends the provision data to Relay

④ LM Emulator Controller calls LM Emulator Communicator to wait TL1 commands response and alarm messages from Relay.

⑤ LM Emulator Communicator receives a message and notifies the LM Emulator Controller

⑥ LM Emulator Communicator writes the alarm message to log file, write the TL1 commands provision data and response result to log file.

3.3.3 GCC and LM Emulator Implementation

GCC Emulator and LM Emulator will be integrated in liblmgcc.so after they are compiled. The directory of GCC Emulator and LM Emulator codes is test_harness/lmgcc. The source files are as follows:

gcc_recv.c : this file is responsible for receiving messages from SIPA and initializing GCC Emulator.
gcc_send.c: this file is responsible for sending messages to SIPA.
logmsg.c: this file is responsible for printing logs/errors in lmgcc.log/lmgcc.err files.
lm.c: this file is responsible for sending provision data to SIPA and handling alarm.
lm_utils.c: this file includes all the functions that lm.c needed.
lm_inc.h: this is the header file for LM Emulator and SWIG.
alarm_msg.c: this file is used for writing alarm information to log file.
tool_utils.c: this file supplies some utilities for GCC Emulator and LM Emulator.
lmgcc.h: header file for GCC Emulator and LM Emulator.
common.h: header file for GCC Emulator and LM Emulator.
lmgcc.i: this file is supplied for SWIG.
makefile: this file will create lmgcc_wrap.c , which is created by SWIG according to lmgcc.i, and liblmgcc.so.

GCC and LM Emulator Code Strategy

1) Use Plexus product header files with least changes on them
2) Create GCC and LM Emulator specific header files, eg. lmgcc.h
3) Keep Plexus log and error messages format
4) Use same code styles as Plexus product
5) For SWIG, GCC and LM Emulator creates its own header files according to Plexus to avoid impact on Plexus product. These header files will be removed as soon as liblmgcc.so is created.

How to create lmgcc.i

This file is automatically generated according to lmgcc.src.i, which is created by tool developers. Lmgcc.src.i will include the header files, global variables and functions that can be used by TCL. Pre-hdl.sh can be used to produce the lmgcc.i according to lmgcc.src.i.

3.4 SIPP [Katie Zhang, Jonathan Li]

3.4.1 Overview

SIPp is a performance testing tool for the SIP protocol. It allows generating one or many SIP calls to one remote system. These calls are established and released by “INVITE” and “BYE” messages. It runs on either Linux or Window. SIP Harness Tool adopts the Linux version.

SIPp may perform as server or client. When taking the role of server, it’s the terminating party in the call scenario; otherwise, it’s originating party.

There are four transport modes in SIPp: UDP mono socket, default in SIPp, is the most general mode. It opens one socket for all calls; UDP multi socket, every call obtains its own IP/UDP socket connection; the other two are TCP mono socket and TCP multi socket, they’re similar to corresponding UDP modes except that their data transfer modes are TCP.

3.4.2 Scenario files

The outgoing SIPp messages are generated according to user’s XML scripts. A XML script file describes a subscriber’s behaviors. It always begins with

<? xml version=”1.0” encoding=”ISO-8859-1” ?>

<scenario name= “Basic Sipstone UAC”>

And end with:

</scenario>

User case is encapsulated in this token. It may contain some elements behaviors like “send messge” and “receive messge”. Here are details:

1. <send>…..</send>
   Send a SIP message to the remote side. The user must fill this token with SIP message.

2. <recv>…..</recv>
   Wait for an incoming message. The user must fill this token with expected message type like “100 TYRING”, “200 OK”.

3. <sendCmd>……</sendCmd>
   Only be used in 3PCC. Commands may be sent to twin 3PCC SIPp instance through this token.

4. <recvCmd>……</recvCmd>
   Only be used in 3PCC. Wait for the message sent by “sendCmd” token.

5. <pause>……</pause>
   Stop in a pointed period.

6. <ResponseTimeRespartition>……</ResponseTimeRespartition>
   Specify the intervals, in milliseconds, used to distribute the values of response times.

7. <CallLengthRepartition>……</CallLengthRepartition>
   Specify the intervals, in milliseconds, used to distribute the values of the call length measures.

A functional scenario should contain two subscribers: one is originating party and the other is terminating party. In other words, a functional scenario simulated completely by SIPp should contain two SIPp instances. Chapter 5 gives a SIPp script example.

SIPp maintains some keywords to represent some system values and user parameters. For example, [service] means service name string, [remote_ip] means IP address of remote side.

For more details, please refer to the manual.

3.4.3 Logs

User can get the SIP message details and call statistics from SIPp logs. They are always named as “<scenario_name>_<pid>_<log_type>.log”. You can find them in work directory. Generally, there are three types of logs: message logs record the sent and received SIP messages; error logs record the error information; screen logs record the call statistics.

3.4.4 Integration with SIP Test Harness

Now there are two available SIPp versions in Linux environment - 1.0 and 1.1. From the perspective of functionality, both are able to generate basic SIP messages. But SIPp 1.1 is an unstable version until now. Therefore, we choose SIPp 1.0, a stable SIPp version which can fully support our application.

SIP test Harness tool implements SIPp as the component of SIP message generator. In call scenario, it takes the role of originating/terminating party. For the sake of user's convenience, harness tool supports

half-call model, which simulates the communication between GCC and orignating/terminating party.

This design benefits tool users in these aspects:

1. The model is more generic. It can support various kinds of complex scenarios by dividing the call flow to multiple GCC-SIP flows. For example, to test 3PCC, users can break the scenario into three GCC-SIP pieces instead of displaying the call flow as a whole.
2. It helps to get rid of complicated GCC configuration. However, tool users are required to have related GCC configuration knowledge - harness tool only simplifies the GCC component instead of removing it.
3. It's easier to display rainy-day call scenario through half-call model.

Another topic is how many SIPp instances will be created in the harness tool. Because the tool implements half-call model, one SIPp instance is enough for most of simple call flows. However, in some complex call scenarios like conference call, multiple SIPp instances are inevitably required.

Harness tool also needs to collect call information from SIPp logs. It includes SIP messages details, call flow, call result and errors. Harness tool will judge whether the call is successful or not from them. These data are also provided to users for further analysis.

One major SIPp feature is call traffic control. Users can configure how many calls they totally want to make and how much the call rate is. However, there is another component realized these features in harness tool. So we don't provide call traffic control through SIPp.

SIPp doesn’t provide mechanisms to deal with message loss. If a SIP message is missed in the transmission, the SIPp will block until user press “q” to exit.

In harness, SIPp is explicitly generated as a child process in “.tc” script through “startupSIPp” method. The start parameters are also configured here. Users could get its output display through telnet to specific TCP port. Of course, you can control it through this interface.

3.4.5 SIPT support [Joey Zhang]

SIPp is mainly used for SIP test, but it could also be used for SIPT test through modifying the SIPp code. Since the SIPp code use many string functions to handle the SIP message, so we couldn’t use it to send ISUP raw data.

So we modifying the code and use the function str_to_hex(char *src, char *dst) to transform the ascii raw data to hex format. After that, we need recalculate the content length and the message length with the function sipt_msg_trans(char *msg).

3.5 Logging Library [Raven Bi, William Cao]

This section covers how we port Plexus logging library (`libTelicaLog.a`) from Lynx to Linux. `LibTelicaLog.a` is used by all Plexus components (GCC, SIPA, and so on) to generate debugging logs. In test harness we still need it (by SIPA and Relay) since we want to simulate the real switch environment better.

Totally there are 7 files that we need to port. They are,

```
logUtils.c
mv_logMgmnt.c
mv_mplogUtils.c
klogthread.c
vsn_printf.c
procMem.c
```

`logCommon.c`

As described in section 4.14, we will use ‘LINUX’ and ‘TEST_HARNESS’ macros for our code changes. Below are more information about the logging stuff porting (This is not going to give a full list but some typical changes that we will make to port logging library to Linux).

1. Create a new makefile named `log.mk` in directory `telica_common/Simulator` which will be used to build all above 7 files to `libTelicaLog.a`.

2. To resolve pthread related APIs.

Change ‘`#ifdef ___LYNXOS4`’ to ‘`#if (defined ___LYNXOS4) || (defined LINUX)`’ since LynxOS 4 thread APIs are POSIX compliant.

3. Use new SEM APIs.

Use sem_wait(), sem_post(), sem_init to replace csem_wait(), csem_signal(), csem_create().

4. To get thread identification, add following macro,

```
#define  Gettid()      ((pid_t)(syscall(__NR_gettid)))
```

5. For some header files, we have to change the incluced path, for example,

change ‘`#include <timeb.h>`’ to ‘`#include <sys/timeb.h>`’.

3.6 SIPA [Raven Bi]

Plexus platform uses Lynx as its operating system for SIPA but the sip test harness is supposed to run on Linux. Then porting SIPA from Lynx to Linux is necessary. The guideline of the porting task is to change as little code as possible so that test harness could simulate the real running environment better.

SIPA will run as a separate process on Linux. The porting work will be divided into three parts based on SIPA constitution – DCL, SIPA and Support Functions.

3.6.1 Makefile

We plan to use existing SIPA makefiles so a new flag – SWTH is added for test harness build so that we will not break Lynx build. Another reason to add new flag to existing makefiles instead of creating new makefiles is that we want to support both DEBUG and RELEASE build of sipa (save efforts).

- Don’t include telica_defs.mk in signaling/sip2.3/jobs/gnu/makefile.

```
ifndef SWTH
include $(TOP)/build/telica_defs.mk
 endif
```

- Make a new copy of file ‘build/telica_rules.mk’ and save it in ‘test_harness/telica_rules.mk’. In that file ew will bypass the checking of ENV_PREFIX variable.

```
  ifndef SWTH
  ENV_PREFIX_BASENAME := $(shell basename $(ENV_PREFIX))
  TOPLYNX_INODE := $(shell ls -lLdi $(TOP)/components/$(ENV_PREFIX_BASENAME)
| sed -e 's/ .*//')
  ENV_PREFIX_INODE := $(shell ls -lLdi $(ENV_PREFIX) | sed -e 's/ .*//')
```

```
ifneq ($(TOPLYNX_INODE), $(ENV_PREFIX_INODE))
include ERROR:current_directory_does_not_match_ENV_PREFIX
endif
endif
```

- Change some makefiles in 'signaling/sip2.3/jobs/gnu' directory to add '`LINUX`' and '`TEST_HARNESS`' definition. For example, in sipt.mak, we will add following,

```
ifdef SWTH
EXTRA += -DLINUX_PORT -DSIMULATOR_TOOL -DSUNOS
Endif
```

- Use a new object directory instead of 'siptdbg' since we may need to build Linux and Lynx version SIPA at the same time. Makefile sipt.mak should be updated to support this.

Please be noted that we used `SUNOS` macro here since some stuff in `telica_common` directory have been ported to SUN OS (use `SUNOS` macro) which is more similar to Linux than Lynx and we could reuse them to save efforts. In fact while porting logging library to Linux we also used `SUNOS` macro (section 4.6). So far no issue was found and everything works well.

3.6.2 DCL

DCL SIP stack is a third party SIP stack that could support many platforms, including both Lynx and Linux. And it provided Linux compile options for us to build on Linux platform. Ideally what we need is only to change DCL makefile option to Linux. But there may be some Plexus added code there and need our modification. That is supposed to be minor and will not be listed here.

- Create a new file named linuxux.cfg

```
#############################################################################
# Makefile:  linuxux.cfg                                                    #
#                                                                            #
# Purpose:   Header file containing all host specific commands for building  #
#            Linux objects on a Unix host.                                  #
#                                                                            #
# (C) COPYRIGHT DATA CONNECTION LIMITED 2001                                 #
#                                                                            #
# $Revision: 1.1.1.1 $$Modtime::   Jun 20 2001 16:22:26   $ #
#                                                                            #
#############################################################################

#############################################################################
# Set the variable to check the object directory exists and if it does not  #
# then create it. The command is dependant on the target architecture, a Unix #
# host will require a Unix version whilst a PC host will require the PC     #
# version.                                                                   #
#############################################################################
CHECK_DIR_EXIST = if [ ! -d $(OBJDIR) ];\
                    then \
                      mkdir $(OBJDIR) ;\
                  fi ;
```

- Add following to linux.cfg

```
HOST_TYPE = ux
include $(OS_TYPE)$(HOST_TYPE).cfg
```

- Change build option to 'linux' in dcinit.cfg

```
# Define the OS_TYPE as one of:
# - vxworks
# - sparc
# - linux
# - ose
# - lynxos
# - sparcmt
# - unixware
```

```
# Sip23 Telica Changes
ifdef SWTH
OS_TYPE = linux
else
OS_TYPE = lynxos
endif
```

In ‘test_harness/bin’ directory we also put a copy of nbase.ini for user to fully control SIPA. That nbase.ini is a little different from Lynx one.

```
trace_file  = ./sip_inttrc
ips_file    = ./sip_ipstrc
pd_file     = ./sip_pdtrc
per_file    = ./sip_pertrc
mem_file    = ./sip_memtrc
```

3.6.3 TELICA_SIPA

For Plexus added code in directory `signaling/sip2.3/tools/telica_sipa`, there is supposed to be not too much OS specific functions calls besides following. They should be changed to corresponding Linux OS APIs.

```
pthread_attr_create
pthread_attr_setschedpolicy
pthread_attr_setprio
pthread_attr_setinheritsched
pthread_attr_setsched
pthread_create
st_name
……
```

Those functions defined in sipsigdbg.c are used for debug purpose. They will also be ported. Some commands are closely tied to the OS (for example sipaDumpProcInfo) and may need code changes. Some other functions (for example those for logging level control) are supposed to work directly.

For PSIF-SIP component located in directory `signaling/sip/psif-sip`, we will not port them since PSIF-SIP is built in GCC process.

To some extent sip test harness is running in integrated environment (see architecture diagram in section 3.1). So regarding some functionalities about distributed environment,

- SIP Redirect

  We will not use it.

- SFEP (SIP Front End Processor)

  That functionality is implemented besides SIPA and from the view of SIPA actions are all almost the same whether it is in integrated or distributed environment except listening port which could be configured using startup arguments (see below).

In test harness we also need a mechanism to let SIPA terminated gracefully (for example, to support gcov, valgrind, etc.). The proposal is to change global variable ‘gEmfActive’ to FALSE via a sigdbg command. That command is hided from user and worked as an internal mechanism in test harness.

> TL1 command syntax: sigdbg:::::dest-a-ccs-slave sipa shutdown s;
>
> Code changes should be in sipa function sipaShutdown().

Another important thing need to clarify is that in sip test harness we will get 1) IP address 2) SIPA listening port from sipa start-up arguments. In Lynx implementation those were input from TL1. From

sipa startup arguments we will also get port numbers for Nbase Relay. For the detailed usage of those two ports, please refer to section 2.4. So to start Linux version SIPA used for test harness, we should use following syntax,

sipa <sipa_listen_ip_addr> <sipa_listen_port> <rly_local_port> <rly_remote_port>

<rly_rep_port> <rly_rep_local_listen_port> <rly_rep_peer_listen_port><slot_id>

<sipa_listen_ip_addr> - The IP address of the test machine

<sipa_listen_port> - On which port will sipa listen on (for sip messages)

<rly_local_port> - NBASE relay port used to communicate with Mtss relay

<rly_remote_port> - MTSS relay port

<rly_rep_port> - Client port in last section

<rly_rep_local_listen_port> - Listening port 1or2 in last section

<rly_rep_peer_listen_port> - Listening port 1or2 in last section

<slot_id> - SP_A_SL (30) or SP_B_SL(31)

<active_standby> - Whether this SIPA is active

<log_directory> - The directory where SIPA logs and core will be stored

3.6.4 Support Functions

Following support functions (located in directory `telica_common`) will be built into libraries and be used by SIPA.

3.6.4.1 Logging

DCL functions used to generate DCL logs are supposed to work on Linux after re-compiling.

For other parts we are calling logging functions in libTelicaLog.a which will also be ported to Linux (see section 3.7).

3.6.4.2 DNS resolution

Most of the DNS resolution functions used by SIPA could be found in Linux library libresolver.a. So on Linux what we need to do is to link Linux resolver library. For those we could not find in Linux libresolver.a,

1) res_reinit(). This function was added by plexus to support ED-DNS-SYS command. To support ED-DNS-SYS on Linux, we have to add following code to function `sipaProcessConfigCmd()`.

```
 case SIP_ED_DNS:
#ifdef LINUX_PORT
        _res.options &= ~RES_INIT;
#else
        res_reinit();
#endif
```

What's more, LM emulator should be able to change resolv.conf according to SIPA configuration file (ED-DNS-SYS). Please refer to section 5.1.4 for more details.

2) On 5.3 we have added some other functions to Lynx resolver to support DNS PM TL1 commands and DNS alarms. That functionality will not be supported in sip test harness and we don't need to change Linux resolver source code.

3.6.4.3 HAPI and other libraries

- Libhapi.a As we could see in section 2.4, we do not use GoAhead although SIPA replication is supported, then it will not be ported.
- Libshmresource.a It will not be used if we choose Linux resolver and don't support `rpastats` SIGDBG command which is used to dump resolver port administrator statistics.
- Libproc.a/Libcmproc.a/Libutil.a The functions in these three libraries are all related to EMF, replication and CM/SP checking. So far we will not port them and just force those SIPA functions that are calling libproc/libcmproc/libutil to return fixed values. For example,

  SipaIsStandbyCold() will call function appGetCpuState() in libcmproc.a. We just remove appGetCpuState and let sipaIsStandbyCold() return FALSE since we have only 1 active SIPA.

3.7 Controller – GCC Emulator Interface [Alex Cao]

The interface between controller and GCC Emulator can accomplish the following functions: initializes the Relay for GCC Emulator, configures Relay channel for GCC Emulator, receives messages from Relay and sends messages to Relay. Detail data structure information can check section 3.12.

3.7.1 Functions for Initializing/Configuring Relay

Before GCC Emulator can communicate with SIPA using Relay, two functions will be called first to initialize and configure Relay, one is SSetProcId and the other is ryCfgChanTCP. The following are detail information:

1. Call SSetProcId function to set process id. SsetProcId prototype:

   PUBLIC Void SSetProcId (procId)

   ProcId will be got from the section 3.3 file.

2. Call RyCfgChanTCP function to configure TCP relay channel. RyCfgChan prototype:

   PUBLIC Void ryCfgChanTCP

   (

   U8 chanId,

   U16 chanType,

   U32 localPortNo,

   U32 remPortNo,

   U8 *remHost,

   ProcId dstProcIdLow,

   ProcId dstProcIdHi,

   ProcId rProcId

)

Expect Controller will call the following function to configure GCC Emulator channel.

```
ryCfgChanTCP(RY_CCS_SIP_CHAN, LRY_CT_TCP_SERVER, 0,
  RY_SIP_CLIENT_PORT(instId),
```

xmitToHostName,

dstProcIdLow, dstProcIdHi, remProcId,

tsmcRyHostHostTxTmr, 10 * tsmcRyHostHostRxTmr);

3.7.2 Functions for Communicating with SIPA

Refer to lmgcc.src.i file.

1. Send messages

   The functions to send messages to SIPA from GCC Emulator. Their definitions are as following:

   1) PUBLIC S16 sendConEvnt (SipwLiConEvnt *sipwLiConEvnt, U32 type)

      This function is used to send connection event to SIPA, including connection request and connection response primitives.

      sipwLiConEvnt is the pointer to messages to be sent.

      type is primitive type.

   2) PUBLIC S16 sendCnStEvnt (SipwLiCnStEvnt *sipwLiCnStEvnt, U32 type)

      This function is used to send connection status request event to SIPA.

      sipwLiCnStEvnt is the pointer to messages to be sent.

      type is primitive type.

   3) PUBLIC S16 sendRelEvnt (SipwLiRelEvnt *sipwLiRelEvnt, U32 type)

      This function is used to send release event to SIPA, including release request and release response primitives.

      sipwLiRelEvnt is the pointer to messages to be sent.

      type is primitive type.

   4) PUBLIC S16 sendSvcEvnt (SipwLiSvcEvnt *sipwLiSvcEvnt, U32 type)

      This function is used to send service request primitive to SIPA.

      sipwLiSvcEvnt is the pointer to messages to be sent.

      type is primitive type.

   5) PUBLIC S16 sendNcEvnt (SipwLiNcEvnt * sipwLiNcEvnt, U32 type)

      This function is used to send non-call primitive to SIPA.

      sipwLiNcEvnt is the pointer to messages to be sent.

      type is primitive type.

   6) PUBLIC S16 sendAudReqEvnt (SipwLiAudReqEvnt *sipwLiAudReqEvnt, U32 type)

      This function is used to send service request primitive to SIPA.

      sipwLiAudReqEvnt is the pointer to messages to be sent.

      type is primitive type.

   7) PUBLIC S16 sendAudCfmEvnt (SipwLiAudCfmEvnt *sipwLiAudCfmEvnt, U32 type)

      This function is used to send service request primitive to SIPA.

      sipwLiAudCfmEvnt is the pointer to messages to be sent.

      type is primitive type.

8) PUBLIC S16 sendGeoCreEvnt (SipwLiGeoCreateEvnt *sipwLiGeoCreEvnt, U32 type)

This function is used to send service request primitive to SIPA.

sipwLiGeoCreEvnt is the pointer to messages to be sent.

type is primitive type.

9) PUBLIC S16 sendGeoAudReqEvnt (SipwLiGeoAudStChgReqEvnt *sipwLiGeoAudReqEvnt, U32 type)

This function is used to send service request primitive to SIPA.

sipwLiGeoAudReqEvnt is the pointer to messages to be sent.

type is primitive type.

2. Receive messages

SRegTTsk will register message receiving handler function, sipwActvTskNew and sipwActvInit. Then it can be used get the messages from SIPA.

```
S16 SRegTTsk
    (
    Ent ent,              /* entity */
    Inst inst,            /* instance */
    Ttype type,           /* task type */
    Prior prior,          /* task priority */
    PAIFS16 initTsk,      /* initialization function */
    ActvTsk actvTsk       /* activation function */
    )
```

The following function is called to receive messages from Relay.

SRegTTsk(ENTSIPW, 0, TTNORM, 0, sipwActvInit, sipwActvTskNew);

SipwActvTskNew is the handler for receiving messages. It will returned the message type. Message bodies will be stored in global variables. The following variables have been supplied:

SipwLiCnStEvnt *sipwLiCnStInd;

SipwLiCnCfmEvnt *sipwLiConCfm;

SipwLiConEvnt *sipwLiConInd;

SipwLiRelEvnt *sipwLiRelInd;

SipwLiRelEvnt *sipwLiRelCfm;

SipwLiSvcEvnt *sipwLiSvcInd;

SipwLiNcEvnt *sipwLiMsgInd;

SipwLiNcEvnt *sipwLiMsgCfm;

SipwLiAudCfmEvnt *sipwLiAudCfm;

SipwLiGeoCreateCfmEvnt *sipwLiGeoCrtCfm;

SipwLiGeoAudStChgIndEvnt *sipwLiGeoAudInd;

3.8 Controller – LM Interface [Joey Zhang]

This interface mainly includes two items, one is used to assign values for configuration data and the other is used to send configuration data to SIPA. Below are the details about the functions.

3.8.1 Data Structure for Configuration

Below are the data structures that will be used in the functions. Since it will need many data structures in the function, for these data structures SiptCfg , SipTrkGrp , SipRoute , SipTgMapAddr , PrflSip , PrflSipT, SigdbgPk, AagSys, you can refer to section 4.11.1.

General data structure:

```
/* LM TL1 event structures */
   enum LmTl1EvtEnums_t
   {
      LM_TL1_ENT_EVT=1,
      LM_TL1_ED_EVT,
      LM_TL1_DLT_EVT,
      LM_TL1_RTRV_EVT,
   };

/* support for sipa failover */
enum Dest_sipa {
    BOTH_SIPA = 0,
    ACTIVE_SIPA,
    STANDBY_SIPA
};

/* sipSys context define */
typedef struct sipgnGenCb
{
   SiptCfg    *siptCfg;
   SiptAdr    *siptAdr;
   U8 lmTl1Evt;
   U8 dest_sipa;
} SipgnGenCb;

typedef struct siptAdr
{
   int     tableId;
/*    int     adrIdx; */  /* Unused for now */
   TkU32   transport;
/*    TkU32   ipAddr; */  /* Unused for now */
   TkU32   port;
/*    TkU32   capacity; */ /* Unused for now */
} SiptAdr;

/* DNSSYS context define */
typedef struct dnsSysCb
{
   DnsSys          dnsSys;
   U8 lmTl1Evt;
   U8 dest_sipa;
} DnsSysCb;

typedef struct dnsSys
{
   int             tableId;
   TkStr64         dnsName;
```

```
   TkStr32        dnsIP1;
   TkStr32        dnsIP2;
   TkStr32        dnsIP3;
   TkS32          dnsStaPortRange;
   TkU8           timeOut;
   TkU8           retry;
   TkU8           numPorts;
} DnsSys;

/* trunkgroup context define */
typedef struct tgpCb
{
   SipTrkGrp   sipTrkGrp;
    U8   lmTl1Evt;
    U8 dest_sipa;
} TgpCb;

/* SIPROUTE context define */
typedef struct sipRouteCb
{
   SipRoute   *sipRoute;
   U8 lmTl1Evt;
   U8 dest_sipa;
} SipRouteCb;

/* SIPTGMAPADDR context define */
typedef struct sipTgMapAddrCb
{
   SipTgMapAddr   sipTgMapAddr;
   U8 lmTl1Evt;
   U8 dest_sipa;
} SipTgMapAddrCb;

/* profileSip context define */
typedef struct prflSipCb
{
   PrflSip    prflSip;
   U8 lmTl1Evt;
   U8 dest_sipa;
} PrflSipCb;

/* siptprofile context define */
typedef struct prflSipTCb
{
  PrflSipT prflSipT;
  U8 lmTl1Evt;
  U8 dest_sipa;
} PrflSipTCb;

/* sipa debug command structure */
typedef struct sigdbgPk
{
   Header hdr;
   U16    index;
   char   usrCmd[SIG_DBG_MAX];
} SigdbgPk;
```

```
/* sipa debug command structure*/
typedef struct sigdbgPkCb
{
   SigdbgPk sigdbgPk;
   U8 dest_sipa;
} SigdbgPkCb;
```

3.8.2 Functions for Assembling Configuration Data

Below are the functions we provide as the interface for tcl to assign the value for each structure.

1) public S16 LmSipgnSysCfg(SipgnGenCb *context) {}

This function is used to assemble the SiptCfg data structure when entering the command ED-SIP-SYS.

2) public S16 LmDnsSysSipaCfg (DnsSysCb *context) {}

This function is used to assemble the DnsSys data structure when entering the command ED-DNS-SYS.

3) public S16 LmSipTrkGrpCfg (TgpCb *cb) {}

This function is used to assemble the SipTrkGrp data structure when entering the command ENT/ED/DLT-TRKGRP.

4) public S16 LmSipRouteCfg (SipRouteCb *context) {}

This function is used to assemble the SipRoute data structure when entering the command ENT/ED/DLT-SIP-ROUTE.

5) public S16 LmSipTgMapAddrCfg (SipTgMapAddrCb *context) {}

This function is used to assemble the SipTgMapAddr data structure when entering the command ENT/ED/DLT-SIP-TGMAPADDR.

6) public S16 LmSipPrflDataCfg (PrflSipCb *context) {}

This function is used to assemble the SipPrflData data structure when entering the command ENT/ED/DLT-PRFL-SIP.

7) public S16 LmSipaDbgCfg (SigdbgPk *context) {}

This function is used to assemble the SigdbgPk data structure when entering the command SIGDBG.

8) public S16 LmAagSysSipaCfg(AagSysCb * context) {}

This function is used to assemble the AagSysCb data structure when entering the command SIP_ED_AAGSYS.

3.8.3 Functions for Sending Configuration Data to SIPA

Below are the functions we provide for sending configuration data to SIPA.

1) public S16 LmSipgnSysSnd(SipgnGenCb *context) {}

This function is used to send the SiptCfg data structure to SIPA.

2) public S16 LmDnsSysSipaSnd (DnsSysCb *context) {}

This function is used to send the DnsSys data structure to SIPA.

3) public S16 LmSipTrkGrpSnd (TgpCb *cb) {}

This function is used to send the SipTrkGrp data structure to SIPA.

4) public S16 LmSipRouteSnd(SipRouteCb *context) {}

This function is used to send the SipRoute data structure to SIPA.

5) public S16 LmSipTgMapAddrSnd (SipTgMapAddrCb *context) {}

This function is used to send the SipTgMapAddr data structure to SIPA.

6) public S16 LmSipPrflDataSnd (PrflSipCb *context) {}

This function is used to send the SipPrflData data structure to SIPA.

7) public S16 LmSipaDbgSnd (SigdbgPk *context) {}

This function is used to send the SigdbgPk data structure to SIPA.

8) public S16 LmAagSysSipaSnd (AagSysCb * context) {}

This function is used to send the AagSysCb data structure to SIPA.

3.9 Controller – SIPP Interface [Jonathan Li]

Expect controller will send the SIPp startup command with arguments, check the SIPp return value and get the testing results from standard out or files and logs.

1. SIPp startup commands as following:
    a. sipp -d 10000 -r 1 -rp 1000 -m 1 -sf uac.xml -inf database.csv -p 5061 135.252.129.38 -trace_err -trace_msg -trace_stat -trace_screen –stf statistics.txt

    -d: Controls the length (in milliseconds) of calls. More precisely, this controls the duration of 'pause' instructions in the scenario, if they do not have a 'milliseconds' section. Default value is 0.

    -r: Set the call rate (in calls per seconds).

    -rp: Specify the rate period in milliseconds for the call rate.

    -m: top the test and exit when 'calls' calls are processed.

    -sf: Loads an alternate xml scenario file.

-inf: Inject values from an external CSV file during calls into the scenarios.

-p: Set the local port number.

-trace_err: Trace all unexpected messages in <scenario file name>_<ppid>_errors.log.

-trace_msg: Displays sent and received SIP messages in <scenario file name>_<ppid>_messages.log

-trace_stat: Dumps all statistics in <scenario_name>_<ppid>.csv> file.

-trace_screen: Dump statistic screens in the <scenario_name>_<ppid>_screens.log file when quitting SIPp.

-stf: Set the file name to use to dump statistics.

135.252.129.38: it can be any IP address which SIPp will send messages to.

b. sipp -sf uas.xml -trace_err -trace_msg -rsa 135.252.129.38:5061

-rsa: Set the remote sending address to host:port for sending the messages.

Other options are same as above.

Detail information about SIPp, you can check http://sipp.sourceforge.net .

2. SIPp has 4 return values: 0, 1, 99 and –1.
   a. 0: All calls were successful
   b. 1: At least one call failed
   c. 99: Normal exit without calls processed
   d. -1: Fatal error
3. SIPp can give out the testing results from standard out or files

   SIPp can use -trace_screen and -trace_stat options to catch the standard out to files, which is helpful when SIPp is running in background.
4. SIPp can print logs/errors in files

   SIPp can use -trace_msg, -trace_err and -trace_timeout to print logs, errors and timeout calls.
5. Controller will also have validation check on SIPp scenario file since SIPp does not have any validation check about the scenario file.

3.10 GCC – SIPA Interface [Alex Cao]

GCC Emulator will call the function SPstTsk to send the messages to SIPA and SregActvTsk to get messages from SIPA, which are defined in Relay.

1. GCC Emulator sends messges

Refer to gcc_send.c file.

SPstTsk definition can be found in tssstub.c, it is as following:

S16 SPstTsk(Pst *pst, Buffer *mBuf)
Pst is used to describe message source, message destination and event etc. Its event needs input from its caller.
MBuffer is message sent to SIPA. Its needs input from its caller.

SipwLiSipPostEvnt calls the SPstTsk in sipw_ptli.c, its definition as following:

S16 sipwLiSipPostEvnt(Data *event, U32 type, U32 dataSize)

Event will be packaged into mBuf, and it's the primitive between GCC and SIPA.
Type is the primitive type, for example EVTSIPWCONREQ. It will be assigned to pst.event.
DataSize is the event length.

So GCC Emulator will get the primitives from scenario/configuration file and call sipwLiSipPostEvnt to send to SIPA.

2. GCC Emulator receives messages

Refer to gcc_recv.c file.

SRegTTsk definition as following:

```
S16 SRegTTsk
(
Ent ent,                /* entity */
Inst inst,              /* instance */
Ttype type,             /* task type */
Prior prior,            /* task priority */
PAIFS16 initTsk,        /* initialization function */
ActvTsk actvTsk         /* activation function */
)
```

SRegTTsk(ENTSIPW, 0, TTNORM, 0, sipwActvInit, sipwActvTskNew) is called in PSIP-SIP. SipwActvTskNew is registered for PSIP-SIP use, which will pass the received messages to GCC Emulator. SipwActvTskNew is defined as following:

S16 sipwActvTskNew(Pst *pst, Buffer *mBuf)
Event will be un-packaged from mBuf, and it's the primitive between GCC and SIPA.
Type is the primitive type, for example EVTSIPWCONREQ. It will be passed from pst.event.

So GCC Emulator will get the primitives from SIPA.

Currently this tool support all events between SIPA and GCC emulator except congestion indication, their definition is in signaling/sigcom/sipt.h. They are using data structures which defined in signaling/sigcom/sipt.x file.

3.11 LM – SIPA Interface [Joey Zhang, Caleb Chen]

3.11.1 Configuration Data Mapping

All configuration data inputted from TL1 are passed to SIPA via RELAY (RELAY porting will be covered in section 3.2).

Following is the overall structure used to store LM data. Please be noted, some members in this structure may be obsolete (for old TL1 commands not supported) but we keep them here for consistence.

```
typedef struct SIPT_CMD_BUF_TAG
{
 unsigned int cmd;
 unsigned int index;
 enum SIPT_RESULT result;
 union
 {
    SiptCfg          siptCfg;
    SiptAdr          siptAdr;
    SipIpAddr        sipIpAddr; /* dev: SIP-IPADDR add */
    sip_mon_data_t   sipMonData;
    SipTrkGrp sipTrkGrp;
    SipRoute         sipRoute;  /* SIPROUTE */
```

```
    SipCcsStateCntrl  sipCcsStateCntrl[TELICA_MAX_CCSID];
    SipTgMapAddr      sipTgMapAddr;
    SipPrflData       sipPrflData;  /* PRFL-SIP */
    dns_mon_data_t    sipDnsMonData;
  } Data;
} SIPT_CMD_BUF_t;
```

A mapping table between TL1 commands and SIPA/LM structure is also provided. For more information about the TL1 commands used for SIP calls, please refer to TL1 command reference guide.

Table Number 3. Mapping table between TL1 commands and SIPA/LM structure

TL1 Command	'cmd' Value	Structure
ED-SIP-SYS	SIPT_ED_CFG	typedef struct siptCfg { int tableId; TkU32 inviteTmout; TkU32 t1Tmr; TkU32 t2Tmr; TkU32 sipPort; TkU32 sigIpAddr; TkU8 sipaMode; TkU32 pst; TkStr32 anncURL; TkU8 remoteAnncReq; TkU16 sendPrack; TkU32 sesTmr; TkU8 inviteRetry; TkU32 dnsCacheTTL; TkU16 cpuPercent; } SiptCfg;
ED-DNS-SYS	SIP_ED_DNS	N/A
ENT/ED/DLT-TRKGRP	SIP_ENT_TGP SIP_ED-TGP SIP_DLT-TGP	typedef struct sipTrkGrp { int tableId; TkU16 tgn; TkU16 sipPrflId; TkU16 sipTEnable; TkU8 sipPlus; TkU16 sipHrtBtTmr; TkU32 sipSigIp; TkU8 sipPClpId; TkStr64 sipSrcFqdn; TkStr64 sipDstFqdn; TkU32 sipDstFqdnIp; TkU16 sipDstFqdnPort; TkU16 sipDstFqdnTransport; TkU32 sipProxyIp; TkU16 sipProxyPort; TkU8 sipCgpMap; TkStr16 sipRouteName; } SipTrkGrp;
ENT/ED/DLT-PRFL-SIP	SIP_ENT_PRFLSIP SIP_ED_PRFLSIP	typedef struct sipPrflData {

	SIP_DLT_PRFLSIP	int tableId; TkU16 siptPrflId; TkU16 sip3xxHandling; } SipPrflData;
ENT/ED/DLT-SIP-ROUTE	SIP_ENT_ROUTE SIP_ED_ROUTE SIP_DLT_ROUTE	typedef struct sipRoute { int tableId; TkStr16 sipRouteName; TkU16 index; TkU16 type; TkStr64 address; TkU16 port; TkU16 routerType; TkU16 transport; } SipRoute;
ENT/ED/DLT-SIP-TGMAPADDR	SIP_ENT_TGMAPADDR SIP_ED_TGMAPADDR SIP_DLT_TGMAPADDR	typedef struct sipTgMapAddr { int tableId; TkStr64 addr; TkU8 type; TkU32 port; TkU16 tgn; } SipTgMapAddr;
ED-AAG-SYS	SIP_ED_AAGSYS	typedef struct _aagSys { int tableId; int version TkStr64 priCscfAddr; TkStr64 secCsCfAddr; TkS16 regTmPeriod; TkStr64 regFqdn; TkStr64 hostFqdn; TkU16 maxReg; TkU8 maxRegRate; TkU32 regRetryTm1; TkU32 regRetryTm2; TkU8 pst; TkU8 country; }AagSys;
SIGDBG-DBGLVL/IPSTRC/IPSFLUSH/DUMPRAW/STTRC……	N/A	typedef struct sigdbgPkCb { SigdbgPk sigdbgPk; U8 dest_sipa; } SigdbgPkCb;
SW-TOPROTN-EQPT	N/A	N/A
ENT/ED/DLT-PRFL-SIPT	SIP_ENT_PRFLSIP SIP_ED_PRFLSIP SIP_DLT_PRFLSIP	typedef struct prflSipT { int tableId; int version; /* Hot Upgrade Support */ TkU16 siptPrflId; TkU16 maxFwd; TkU16 maxFwdHopCntrRatio; TkU16 pdcsBilling; TkU16 privacy; TkU16 reasonHdr; TkU16 pCharging;

		TkU16 sip3xxHandling; TkU16 cgpnIwProc; TkU16 gnumIwProc; TkU16 sipFciIwProc; TkU16 fciNpdiIwProc; TkU16 sipBciIwProc; TkU16 nciIwProc; TkU16 cancelIwProc; TkU16 byeIwProc; TkU16 sip181IwProc; TkU16 sip182IwProc; TkU16 sip183IwProc; TkU16 causeIwProc; TkU16 cpg456IwProc; TkU16 ciclwProc; TkU16 t7; TkU16 origTrkgrp; } PrflSipT;

Based on current architecture we will not support RTRV and PM functionalities in LM emulator , that is to say, the data structures for following TL1 commands will not be considered. This may changed in the future.

RTRV-SIP-SYS

RTRV-DNS-SYS

RTRV-PRFL-SIP

RTRV-SIP-ROUTE

RTRV-SIP-TGMAPADDR

INIT-REG-SIPMSG

RTRV-PM-SIPMSG

INIT-REG-DNS

RTRV-PM-DNS

3.11.2 Alarm Data

The structures that used to store Alarm Data from SIPA to LM emulator are as following.

```
/*
 * SIP to FAM Event Data Structure
 */
typedef struct SIPEVT_DATA_TAG
{
   int EventType; /* The SIP Event Type (listed above) */

   union
   {
      SIP_FS_FAILOVER_EVT_t      FsFailover;
      SIP_FS_UNAVAIL_EVT_t       FsUnavail;
      SIP_SOCKET_ERR_EVT_t       SocketErr;
      SIP_CONG_THRESHOLD_EVT_t   CongThreshold;
      SIP_DNS_CONNECT_FAIL_EVT_t DnsConnectFail;
      SIP_AAG_423_EVT_t          Aag423Evt;
```

```
    } EventData;
} SIPEVT_DATA_t, sipevt_data_t;
/*********************************************************/
/*
 *  Unsolicited Status (Alarm) Structure
 */
/*********************************************************/
typedef struct SipUnsolicitedStatus_s {
    SIPEVT_DATA_t       data;
} SipUnsolicitedStatus_t;

/*********************************************************/
/*
 *  Overall Structure
 */
/*********************************************************/
typedef struct sipMngmnt {
    Header hdr;
    CmStatus cfm;
    union
    {
        SipUnsolicitedStatus_t unsolicitedStatus;       /* Unsolicited Status */
    } t ;
} SipMngmt;
```

SipMngmt.hdr.msgType will be set to TUSTA and msgLen is sizeof(SipMngmt).

3.11.3 Function for Handling Alarm

For the data structure of alarm data, please refer to section 4.11.2. After calling SRegTTsk function to initialize/register receiving message handler function. Once messages are arrived includes TL1 commands response and alarm message, the callback function will be invoked. Below is the function for handling the messages from SIPA.

S16 lmActvTsk (Pst *pst, Buffer *mBuf) {}

In the function, it will call the alarm handler function if the message is alarm. Below is the function for handling the alarm information.

Public void LmSiptAlarmHndlr (Pst *pst, Buffer *mBuf) {}

3.11.4 Handle TL1 Commands Response

Once the TL1 commands for configuration are sent to SIPA, the user want to see the commands result from the screen. For the current supported TL1 commands, except for the sigdbg commands, all the other TL1 commands have the response message from SIPA which says that the command is successful or failed. When the LM emulator receives the TL1 commands response, if it fails it will print the result to the screen. Below is the function for handling the TL1 commands response from SIPA.

S16 lmActvTsk (Pst *pst, Buffer *mBuf) {}

3.11.5 SIGDBG commands

All SIGDBG commands from TL1 are passed to SIPA via RELAY (RELAY porting will be covered in section 4.2).

Following is the structure used to store SIGDBG commands data.

```
typedef struct sigdbgPk
{
   Header hdr;
   U16    index;
   char   usrCmd[SIG_DBG_MAX];
} SigdbgPk;
```

The ‘usrCmd’ value will be set to the content of SIGDBG commands. For example, if we input ‘sigdbg:::::dest-b-ccs-slave sipa dbglvl 0;’ from TL1 agent, ‘usrCmd’ will be set to ‘dbglvl 0’. For the syntax of all SIGDBG commands for logging level, pelase refer to section 5.1.4.

3.11.6 LM Interface Implementation in SWIG [Caleb Chen]

Usage of SWIG, please refer section 4.1.4. This section focuses on some concrete implementation of LM interface in SWIG.
Generally, there are three mandatory files:<lm.c>, <lm.h>, <lm.i>. lm.i file is the interface file contains ANSI C function prototypes and variable declarations, which you want to access from Tcl. Then we can build the Tcl module using SWIG into a shared library that can be loaded into Tcl. When loaded, Tcl can now access the functions and variables declared in the SWIG interface file as Tcl extension commands.

3.11.7 TL1 command parsing

The parser should support Tl1 commands mentioned in section 4.11.1 and 4.11.3. The main functionality of the parser is: form the structure with the data in the command, return the structure back to LM. The mapping table between Tl1 command and SIPA/LM structure is provided in section 4.11.1.

The parser is implemented in Tcl, and can be wrapped as an extension Tcl command. The follwing is an example about the format of the command:

runtl1 command_string

The parser can be executed in two modes: run single command one by one as interacting mode, and run bulk of commands listed in a script. This can be easily implemented in Tcl because the command “runtl1”is a Tcl command and Tcl supports both these two executive modes.

There are several key issues we have to consider. First, how to form a C structure in Tcl? This is the scope of SWIG. After compiled the LM source files and header files with SWIG, a function named *wrap_new_structurename* will be created. It is used to declare a new C structure in Tcl, and it can be invoked as a Tcl command. For example, the structure *SiptCfg* defined in LM header files will have a according function like the follwing one:
static int
*_wrap_new_SiptCfg(ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]) {}*

And the SWIG also create the functions used to populate and retrieve the relative members. Such as the member *sigIpAddr*, will have two functions: *_wrap_SiptCfg_sigIpAddr_set* used to populating and *_wrap_SiptCfg_sigIpAddr_get* used to retrieving.

Then if you want to declare a new structure, you can run command (the prefix *_wrap* can be omitted) *set ns [new_SiptCfg]* as a Tcl command.

The second issue is how to map the specific data in Tl1 command to its relative field in structure. There are two kinds of mapping. First, field in TL1 command has the relative member in structure. If both the name and value are listed out in command, such as in the form of fieldname1=value1,fieldname2=value2,…., just parse the field name and value out, and populate it; If only value is listed, we should decide which parameter it belongs to according to the input format, and

populate it with the given value. The second kind of mapping is the command doesn't have the relative field in the structure; we need populate it in LM or give it a default value in Tcl. I prefer the later one.

We consider the input format and value are right in syntax; we just do a little syntax check now. This may be improved in future.

To simplify it, here is a figure for the process.

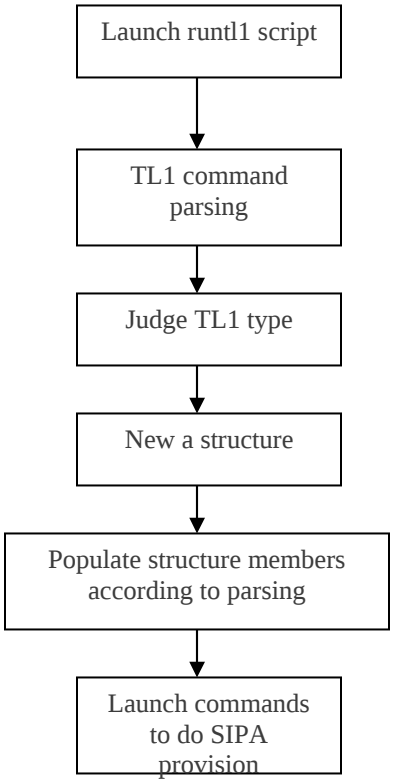


Figure 4.11.6 TL1 parsing process

3.12 SIPA – SIPp Interface [Raven Bi]

SIP messages over IP will be exchanged between SIPA and SIPP although they run on the same OS. This loose coupled and standard interface make it easy for us to update SIPp with newer version or replace SIPp by other SIP message generators. So far Plexus only supports TCP and UDP. In the future if we need to test over the other transport protocols (such as SCTP), SIPp enhancement is needed.

Since SIPp and SIPA will run on the same OS (single IP address allocated), they will use different ports. The allocation of IP Ports will be considered in section 3.3.

4 Test Script

For the test cases description, please refer to the file "_scriptindex.txt" in the script directory, which is `TelicaRoot/components/test_harness/scripts/sip/`.

5 Output [Jonathan]

(Note: this section is planed to be added in later release. So for this moment, we just leave it blank here)

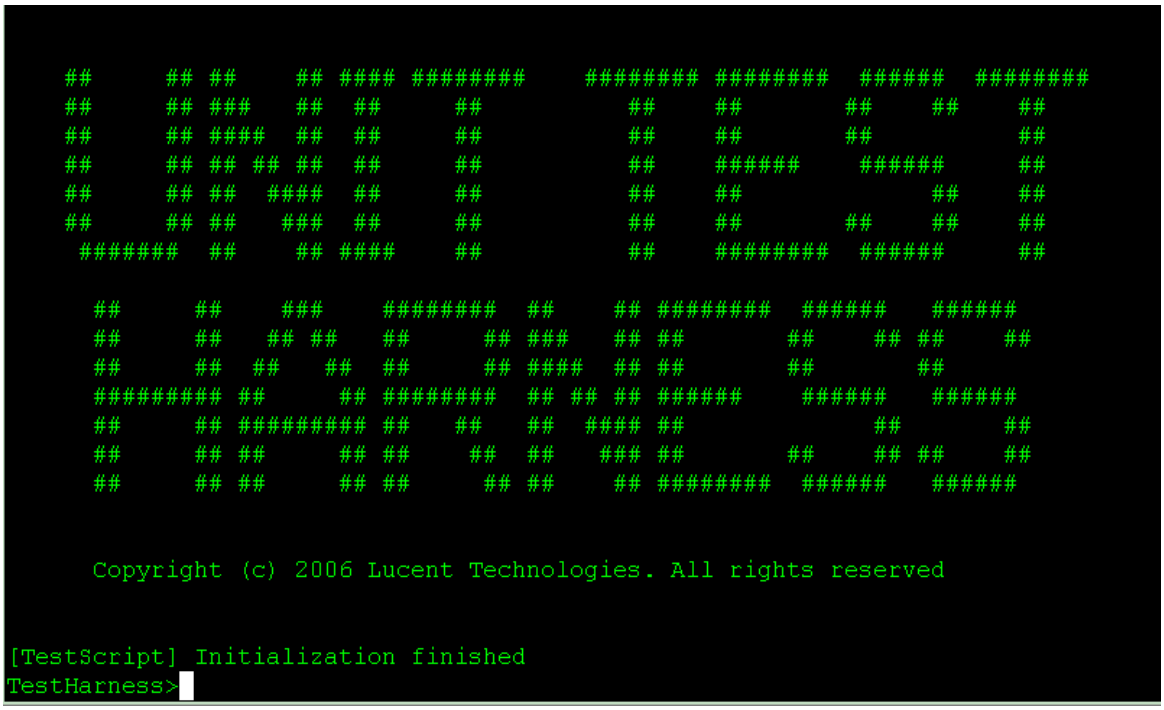
5.1 Example Output Screen, Test Report

5.1.1 Output of Expect Controller

Harness users test the SIP call through writing the configure scripts and operating the Expect controller. So the direct running results all come from the screen display of expect controller. Meanwhile, users can get the test results and running details from the logs for related components.

The output of Expect Controller includes screen display and log record. Expect Controller's log is concern with debug work of developers. For users, the screen display is much more important.

Startup screen:



Command line:

Result Screen:

```
[TestScript] One TestCase Finished!
Test cases (1/1) completed successfully

-----Test Result------
  Total Test Cases: 1
  Completed Test Cases: 1
  Passed Test Cases: 1
  Failed Test Cases: 0
----Completed Test------
  TestCase 1: Script: success; SIPP: success
```

5.1.2 Output from user

In all configuration scripts, users can print their own words in the screen through "print" command. These output all begin with "[TestScript]" to demonstrate that this line is from user's scripts.

5.1.3 LM/GCC Emulator Logs

The debug message of LM/GCC Emulator will be printed in lmgcc.dbg file. And error messages of LM/GCC Emulator will be printed in lmgcc.err file. Both debug information and error information is kept original Plexus log format, which will be helpful for log analysis. And the alarm from SIPA will be outputted to alarm.log file.

5.1.4 Test Report

Harness is not attending to provide a report file to users. The test result will be printed out in the screen in the format like these:

```
[TestScript] One TestCase Finished!
Test cases (1/1) completed successfully

-----Test Result------
  Total Test Cases: 1
  Completed Test Cases: 1
  Passed Test Cases: 1
  Failed Test Cases: 0
----Completed Test------
  TestCase 1: Script: success; SIPP: success
```

5.2 Logging Collection & Parsing

5.2.1 Log Collection

Logging collection is an important topic in Harness. Generally, log types include: SIPA logs, SIPp logs, Relay logs, LM/GCC emulator logs and other logs which are only necessary to harness developers. For each test case, more than ten logs are totally provided to harness users for analysis. Beside that, different module may generate logs in different places. To manage these log files, all logs for a siptest instance are stored in an independent sub-directory under …/test_harness/bin/. The log directory is generated automatically and named according to the date and time of starting siptest, for instance, 1120092300 indicates the log directory for the running of siptest beginning at Nov. 20th 9:23:00 am.

In Alpha version, we don't attend to change the log output path because of time limitation. Users can

find the logs as such routines:

1) Logs for SIPA, SIPp, LM, GCC and ExpectController are placed in the same directory with their corresponding binary file.
2) Log names:

   SIPA: Log names are the same as in the real switch.

   SIPp: <script_name>_<process_id>_<message/error>.log

   LM & GCC: lmgcc.dbg

The SIPA logs maintain the file name and content format accordingly. Users can use “tail –f “ to observe the SIPA output as in the real switch.

[Logs: make as close to the real switch as possible??]

5.2.2 Log Parsing

Log parsing is under design right now.

6 Glossary

Acronym	Description
DCL	Data Connection Limited
DNS	Domain Name Service
GCC	Generic Call Control
LM	Layer Management
TEC	Tcl Extension Commands
SIGDBG	Signaling Debug
SIPA	SIP Agent
SWIG	Simplified Wrapper and Interface Generator
TL1	Transaction Language One
3PCC	3th Party Call Control

7 Appendix A. What plexus code changes will affect test harness [Raven Bi]

Test harness is supposed to evolve with real plexus code but there are some cases in which new plexus changes will break test harness build & testing. This section is to specify what kind of plexus code changes will affect test harness as a reference for test harness users (SIP developers).

7.1 Code changes will affect test harness build

7.1.1 Lynx OS specific API

Test harness is built on Linux platform but our plexus code is originally written for Lynx OS. In test harness some plexus files are reused and they have been ported to Linux. So, if there is any Lynx OS specific APIs newly added to those files (please refer to section 8.1.4 for a complete file list), test harness build will definitely be broken.

For example,

In function sipa_ccl_request_rx(), if we add following code segment to parse 'duration' parameter of SIP messages, test harness build will be broken because catol() is an Lynx OS specific API.

```
if ( OS_STRNCMP(param->name.text,"duration",param->name.length) == 0 )
        {
          NBB_CHAR durationStr[16];
          durationStr[0] = '\0';
          OS_STRNCPY(durationStr, param->value.text, param->value.length);
          minDuration = catol(durationStr);
         SIPA_TRACE (NBB_DETAIL_TRC,
                              ("%s: parsed duration value: %d\n",funcName, minDuration));
          param = (SIPP_HDR *)NBB_NEXT_IN_LIST(param->lqe);
          continue;
        }
```

To fix this kind of issue, just change the Lynx OS specific API to an POSIX one so that the code could be more portable across Unix & Linux world. Another solution is to use 'LINUX' compile flag to wrap the specific APIs if you still want to keep the Lynx OS specific API.

Solution 1: Change to POSIX function

```
if ( OS_STRNCMP(param->name.text,"duration",param->name.length) == 0 )
        {
          NBB_CHAR durationStr[16];
          durationStr[0] = '\0';
          OS_STRNCPY(durationStr, param->value.text, param->value.length);
         minDuration = atoi(durationStr);
          SIPA_TRACE (NBB_DETAIL_TRC,
                              ("%s: parsed duration value: %d\n",funcName, minDuration));
          param = (SIPP_HDR *)NBB_NEXT_IN_LIST(param->lqe);
          continue;
        }
```

Solution 2: Use ‘LINUX’ compile flag

```
if ( OS_STRNCMP(param->name.text,"duration",param->name.length) == 0 )
        {
          NBB_CHAR durationStr[16];
          durationStr[0] = '\0';
          OS_STRNCPY(durationStr, param->value.text, param->value.length);
#ifdef LINUX
          minDuration = atoi(durationStr);
#else
          minDuration = catol(durationStr);
#endif
          SIPA_TRACE (NBB_DETAIL_TRC,
                        ("%s: parsed duration value: %d\n",funcName, minDuration));
          param = (SIPP_HDR *)NBB_NEXT_IN_LIST(param->lqe);
          continue;
        }
```

7.1.2 Header files included by LM_GCC Emulator

Most LM_GCC Emulator functionalities are newly developed in test harness (existing plexus LM and GCC stuff are not reused) but we reused many plexus header. Those header files are mainly used to define some data structures that will be referred by LM_GCC emulator and they are included by LM and GCC stuff in real plexus product. If some changes are newly added to those header files, there will be potential issue. It is hard to define a general rule for this kind of build error but they are all related to code inconsistent. For example,

In LM emulator file lm.c, to use data structure ‘DnsSysCb’, header file lm_inc.h is included since ‘DnsSysCb’ is defined in it. However, if we move the declaration of ‘DnsSysCb’ to another header file, say lm_inc_new.h, lm.c must also be updated to include new header file ‘lm_inc_new.h’, otherwise there will be build error. Please refer to section 8.1.4.4 for the header file name list included by LM_GCC emulator.

7.1.3 Makefile Related

For most test harness components, there are different make files from real plexus code. Then here comes a general rule – if you have updated plexus make files, please make sure test harness make file is also updated if necessary. For example, if we have updated some header files (for example ssi.h) and Lynx make files for a plexus new feature (-Dxxx parameter is added to Lynx make files), test harness build may fail because test harness (LM Emulator) is using a different make file and ssi.h is also included by LM Emulator. There is mismatch between test harness make file and the newly added header file changes. The solution for this kind of issue is to update LM Emulator make file – add –Dxxx parameter to test harness make file either.

Please refer to following for all test harness make files.

TelicaRoot/components/test_harness/makefile - overall make file

TelicaRoot/components/telica_common/Simulator/log.mk - logging library

TelicaRoot/components/signaling/sip2.3/jobs/gnu/*.mak - SIPA make files.

Test Harness specific parts
are wrapped with #ifdef SWTH.

TelicaRoot/components/signaling/mtss/Simulator/Makefile - MTSS make file

TelicaRoot/components/signaling/relay/Simulator/Makefile - Relay make file

TelicaRoot/components/test_harness/expect_controller/Makefile - Expect controller make file

TelicaRoot/components/test_harness/lmgcc/Makefile - lm_gcc emulator make file

7.1.4 Reused Plexus Files

7.1.4.1 SIPA & NBASE Relay

The files in directory 'TelicaRoot/components/signaling/sip2.3' to build SIPA product.

7.1.4.2 MTSS Relay

TelicaRoot/components/signaling/relay/ry_bdy1.c

TelicaRoot/components/signaling/relay/ry_bdy2.c

TelicaRoot/components/signaling/relay/ry_bdy3.c

TelicaRoot/components/signaling/relay/ry_bdy4.c

TelicaRoot/components/signaling/relay/ry_ptli.c

TelicaRoot/components/signaling/relay/ry_id.c

TelicaRoot/components/signaling/relay/ry_ex_ms.c

TelicaRoot/components/signaling/relay/ry_ptmi.c

TelicaRoot/components/signaling/relay/ry_lsb.c

TelicaRoot/components/signaling/relay/smryptmi.c

TelicaRoot/components/signaling/relay/smryexms.c

TelicaRoot/components/signaling/relay/smrybdy1.c

TelicaRoot/components/signaling/relay/ry_tcp.c

TelicaRoot/components/signaling/relay/cm_inet.c

TelicaRoot/components/signaling/relay/cm_gen.c

TelicaRoot/components/signaling/relay/sm_bdy1.c

TelicaRoot/components/signaling/relay/sm_ex_ms.c

TelicaRoot/components/signaling/relay/ry_sim.c

TelicaRoot/components/signaling/mtss/ss_gen.c

TelicaRoot/components/signaling/mtss/ss_task.c

TelicaRoot/components/signaling/mtss/ss_drvr.c

TelicaRoot/components/signaling/mtss/ss_timer.c

TelicaRoot/components/signaling/mtss/ss_mem.c

TelicaRoot/components/signaling/mtss/ss_strm.c

TelicaRoot/components/signaling/mtss/ss_msg.c

TelicaRoot/components/signaling/mtss/ss_queue.c

TelicaRoot/components/signaling/mtss/ss_pack.c

TelicaRoot/components/signaling/mtss/ss_rtr.c

TelicaRoot/components/signaling/mtss/ss_acc.c

TelicaRoot/components/signaling/mtss/cm_mem.c

TelicaRoot/components/signaling/mtss/cm_lib.c

TelicaRoot/components/signaling/mtss/cm_bdy5.c

TelicaRoot/components/signaling/mtss/mt_ss.c

TelicaRoot/components/signaling/mtss/mt_id.c

ANY header files included by above files

7.1.4.3 Logging Library

TelicaRoot/components/telica_common/src/logUtils.c

TelicaRoot/components/telica_common/src/mv_logMgmnt.c

TelicaRoot/components/telica_common/src/mv_mplogUtils.c

TelicaRoot/components/telica_common/src/vsn_printf.c

TelicaRoot/components/telica_common/src/procMem.c

TelicaRoot/components/telica_common/src/logCommon.c

ANY header files included by above files

7.1.4.4 Some files used for LM_GCC emulator

TelicaRoot/components/signaling/sigcom/envopt.h

TelicaRoot/components/signaling/sigcom/envdep.h

TelicaRoot/components/signaling/sigcom/envind.h

TelicaRoot/components/telica_common/inc/TsmColDefs.h

TelicaRoot/components/table_defs/tb.h

TelicaRoot/components/table_defs/tb_sipt.h

TelicaRoot/components/signaling/sigcom/gen.h

TelicaRoot/components/signaling/sigcom/ssi.h

TelicaRoot/components/signaling/sigcom/gen.x

TelicaRoot/components/telica_common/inc/sm_cmds.h

TelicaRoot/components/telica_common/inc/sipt_cmds.h

TelicaRoot/components/signaling/sigcom/cm_mgl.x

TelicaRoot/components/signaling/sigcom/cm_avl.h

TelicaRoot/components/table_defs/tb_cas.h

TelicaRoot/components/table_defs/tb_isup.h

TelicaRoot/components/signaling/sigcom/cm_hash.x

TelicaRoot/components/signaling/sigcom/cm_ss7.h

TelicaRoot/components/signaling/sigcom/cm_atm.h

TelicaRoot/components/signaling/sigcom/cm_cc.h

TelicaRoot/components/signaling/sigcom/int.h

TelicaRoot/components/signaling/sigcom/sipt.h

TelicaRoot/components/signaling/sip/psif-sip/sipw.h

TelicaRoot/components/signaling/sigcom/mgcpt.h

TelicaRoot/components/signaling/tsm/tsmc/git_class.h

TelicaRoot/components/signaling/sigcom/cm_sdp.h

TelicaRoot/components/signaling/sigcom/ssi.x

TelicaRoot/components/signaling/sigcom/cm_ss7.x

TelicaRoot/components/signaling/sigcom/cm_atm.x

TelicaRoot/components/signaling/sigcom/cm_cc.x

TelicaRoot/components/signaling/sigcom/int.x

TelicaRoot/components/signaling/sigcom/sit.x

TelicaRoot/components/signaling/sigcom/sipt.x

TelicaRoot/components/signaling/sigcom/mgct.h

TelicaRoot/components/signaling/sigcom/mgcpt.x

TelicaRoot/components/signaling/sigcom/cm_cas.x

TelicaRoot/components/signaling/sigcom/cct.x

TelicaRoot/components/signaling/sip/psif-sip/sipw.x

TelicaRoot/components/signaling/sigcom/lsip.h

TelicaRoot/components/signaling/sigcom/lsip.x

TelicaRoot/components/signaling/lm/common/lm.h

TelicaRoot/components/signaling/lm/common/lm_sigdbg.h

TelicaRoot/components/table_defs/tb_dnssys.h

TelicaRoot/components/signaling/lm/clam/cl_dnssys.h

TelicaRoot/components/table_defs/tb_rst.h

TelicaRoot/components/signaling/lm/clam/cl_route.h

TelicaRoot/components/signaling/lm/clam/cl_siptgmap.h

TelicaRoot/components/signaling/lm/clam/cl_prflsip.h

TelicaRoot/components/signaling/lm/clam/cl_sipgn.h

TelicaRoot/components/signaling/lm/clam/cl_tgp.h

TelicaRoot/components/signaling/lm/clam/cl_evt.h

TelicaRoot/components/telica_common/inc/sip_evts.h

TelicaRoot/components/signaling/lm/clam/cl.h

TelicaRoot/components/signaling/sigcom/lcs.h

ANY header files included by above files

7.2 Code changes will affect test harness testing

Most test harness broken cases are related to build error. But even if the test harness build is OK, your testing may also be broken if the new code changes are related to following items.

7.2.1 PSIF Primitive Changes

In test harness we have a file named gcc.tcl in which some TCL functions are defined to prepare the PSIF primitive to SIPA. This file will not be evolved automatically with real Plexus code so if your code changes about PSIF primitive structures, gcc.tcl file should also be updated. Otherwise the data send to SIPA may be disordered.

Those PSIF primitives are defined in file TelicaRoot/components/signaling/sigcom/sipt.x, so if you have changed sipt.x, test harness testing may be broken.

7.2.2 TL1 & LM Changes

In test harness TL1 parser and LM emulator are both newly developed. They will not be evolved with real plexus code. So, if you have updated SIPA and LM code to support a new SIPA TL1 command or parameter, to test it in test harness, TL1 parser and LM emulator should be updated to support the new TL1 command.

Here is a brief summary about how to add a new TL1 command in test harness tool.

For the structure and basic concept about TL1 parser, please refer section 3.11.7.

For the Tcl part, there are three .tcl files you have to edit, they are all under the folder “/test_harness/expect_controller/tcl/”:

runtl1.tcl: The main entrance of the TL1 parser.

tl1_glbvar.tcl: Define some common used variables for TL1 parser.

tl1_ utils.tcl: Define some functions to handle every specific TL1 command.

For the runtl1.tcl, it has a procedure named “runtl1” used to check the type of inputted TL1 commands, call the accordingly functions defined in tl1_utils.tcl to handle the commands, and call the relative LM interface functions to do the provision. For wrapping the LM interface functions from C to Tcl, please refer sections of SWIG.

tl1_glbvar.tcl defines some common used variables. There are two kinds of variables. The first kind is some global macros, which are defined in C as enums. For example SIPT_STATE and SIPT_TYPE:

```
enum SIPT_STATE {
   SIPT_IS=0,
   SIPT_OOS,
};
enum SIPT_TYPE {
```

```
  SIPT_UDP=0,
  SIPT_TCP,
  SIPT_BOTH,
};
```

You can use SIPT_IS and SIPT_OOS as Tcl variables after you global them as the following:

```
global SIPT_IS
global SIPT_OOS
global SIPT_UDP
global SIPT_TCP
global SIPT_BOTH
```

The tool reuses these macros to make the program more meaningful, and this can also minimize maintain work. To know more about how to wrap these macros from C part to Tcl part, please refer section of SWIG. The second kind is array used to store the structure members' type. For each TL1 command, there is an accordingly structure used to store the TL1 information and it will be passed to LM. Take ent-trkgrp for example, it has an accordingly structure named sipTrkGrp, who has several members such as tgn, sipProflId, sigType, and so on. Their type are all TKU16, which is also a minor structure.

tl1_utils.tcl defines two kinds of functions. The first one is for handling specific TL1 commands and each command has an accordingly function. Each function takes TL1 command as an inputting argument, and uses regular expression to do the parsing work. All useful information will be parsed out and set to the according structure members. Take ent-trkgrp for example, the regular expression is as following:

```
set pattern {(ent-trkgrp):([0-9a-zA-Z]{0,20}):([0-9]{1,4}):([0-9a-z]{0,6})::
(sip|sipgm|siptnl_kpnv2|siptnl_kpnnl2|siptansi|siptituq767|siptetsiv2|siptin|siptitu1992):([^:]+)}
```

For Tcl regular expression syntax, please refer
http://mobility.ih.lucent.com/~rainbow/tcl/man/tcl_man/regexp.n.html

The second one is for initializing new structures. To add a new command, you don't need to add some basic structure definitions, such as newTkU8, newTkU16. You just need to add the ones that the new adding TL1 command needs.

For the c code, you may refer to the existed plexus code. In lmgcc directory, there are some files related to adding a new TL1 command. Lm_inc.h is the header files for the new added data structures and functions declarations, lm.c is mainly used to implement the provision functions. Let's take ent/ed/dlt-prfl-sipt as an example.

First, you should find the associated data structure for PrflSipT, then add the associated data structure in lm_inc.h as following format.

```
typedef struct prflSipTCb
{
  PrflSipT prflSipT;
  U8 lmTl1Evt;
  U8 dest_sipa;
} PrflSipTCb;
```

Secondly, you should find the associated c code cl_prflsipt2.c. Then in the file, you could write the funciton LmSipTPrflDataSnd according to the existed function clPrflSipTSipCfg. Be careful with the each data item and related operations in the function.

8 References

8.1 Lucent Technologies Documentation

1. PLUS Architecture Specification (79-5017)
2. Plexus Test Harness Architecture Document (79-5027)
3. Plexus PSIF-SIP Software Design (79-3242)
4. Plexus SIP Test Harness User Manual (79-3534)
5. CCT changes for MGC Geographic Redundancy (79-3461)
6. GCC Geographic Redundancy (79-3462)
7. MGC Geographic Redundancy Specification (79-3369)

8.2 Third-party Specifications

1. Tcl manual pages: http://mobility.ih.lucent.com/~rainbow/tcl/man/tcl_man.html.
2. DCL N-BASE Version 2.0 Porting Guide
3. GDB Manuals http://www.gnu.org/software/gdb/documentation/
4. GCC Manuals http://gcc.gnu.org/onlinedocs/
5. SIPp Reference Document 1.1 http://sipp.sourceforge.net/doc1.1/index.html
6. SWIG http://www.swig.org/index.html
7. GCOV http://ggcov.sourceforge.net/
8. Valgrind http://www.valgrind.org/