CRS−1 and IOS XR Operational Best Practices

Document ID: 71770

Contents

Introduction

This document helps you to understand these:

- Process and Threads
- CRS−1 Fabric
- Control Plane
- Rommon and Monlib
- Physical Layer Interface Module (PLIM) and Modular Service Card (MSC)
- Configuration Management
- Security
- Out of Band
- Simple Network Management Protocol (SNMP)

Prerequisites

Requirements

Cisco recommends that you have knowledge of Cisco IOS® XR.

Components Used

The information in this document is based on these software and hardware versions:

- Cisco IOS XR Software
- CRS−1

The information in this document was created from the devices in a specific lab environment. All of the devices used in this document started with a cleared (default) configuration. If your network is live, make sure that you understand the potential impact of any command.

Conventions

Refer to Cisco Technical Tips Conventions for more information on document conventions.

Cisco IOS XR Overview

Cisco IOS XR is designed to scale. The kernel is a Microkernel architecture so it provides only essential services such as process management, scheduling, signals, and timers. All other services such as file systems, drivers, protocol stacks and applications are considered as resource managers and run in memory protected user space. These other services can be added or removed at run time, which depends on the program design. The footprint of Microkernel is only 12 kb. The Microkernel and underlying operating system is from QNX Software Systems, and is called Neutrino. QNX specializes in Real Time Operating System design. The Microkernel is preemptive, and the scheduler is priority based. This ensures that context switching between processes is very fast, and the highest priority threads always have access to CPU when required. These are some of benefits of which Cisco IOS XR takes advantage. But, the biggest benefit is the inherit design of Inter Process Communications within the operating systems core.

Neutrino is a message passing operating system, and messages are the basic means of the Interprocess communications among all threads. When a particular Server wants to provide a service, it creates a channel for exchanging messages. Clients attach to the Servers channel by directly mapping to the relevant file descriptor in order to utilize the service. All communications between client and server is by the same mechanism. This is a huge benefit for a super computer, which CRS−1 is. Consider these when a local read operation is performed on a standard UNIX kernel:

- Software interrupt into the kernel.
- Kernel dispatches into the file system.
- Data is received.

Consider these in the remote case:

- Software interrupt into the kernel.
- Kernel dispatches NFS.
- NFS calls the networking component.
- Remote dispatches the networking component.
- NFS is called.

- Kernel dispatches the file system.

The semantics for the local read and the remote read are not the same. Arguments and parameters for file locking and setting permissions are different.

Consider the QNX local case:

- Software interrupt into the kernel.
- Kernel performs message passing into the file system.

Consider the non local case:

- Software interrupt into the kernel.
- Kernel goes into QNET, which is the IPC transport mechanism.
- QNET goes into kernel.
- Kernel dispatches the file system.

All the semantics that concern argument passing and file system parameters are identical. Everything has been decoupled at the IPC interface that allows the client and server to be completely segregated. This means any process can run anywhere at any point in time. If a particular Route Processor is too busy servicing requests, you can easily migrate those services to a different CPU that runs on a DRP. A super computer that runs different services on different CPUs spread across multiple nodes that can easily communicate with any other node. The infrastructure is in place in order to provide the opportunity to scale. Cisco has utilized this advantage and written additional software that hooks into the principle operations of the message passing kernel that allows the CRS router to scale to thousands of nodes, where a node, in this case a CPU, runs an instance of the OS, whether it is a Route Process (RP), a Distributed Route Processor (DRP), a Modular Services Card (MSC), or a Switch Processor (SP).

Process and Threads

Within the bounds of Cisco IOS XR, a process is a protected area of memory that contains one or more threads. From the programmers perspective, the threads do the work, and each completes a logical execution path in order to perform a specific task. The memory the threads require during the flow of execution belongs to the process they operate within, protected from any other processes threads. A thread is a unit of execution, with an execution context that includes a stack and registers. A process is a group of threads that share a virtual address space, although a process can contain a single thread but more often contains more. If another thread in a different process attempts to write to the memory in your process, the offending process is killed. If there is more than one thread that operates within your process, then that thread has access to the same memory within your process, and as a result is capable to overwrite the data of another thread. Complete the steps in a procedure in order to maintain synchronization to resources in order to prevent this thread within the same process.

A thread uses an object called a Mutual Exclusion (MUTEX) in order to ensure mutual exclusion to services. The thread that has the MUTEX is the thread that can write to a particular area of memory as an example. Other threads that do not have the MUTEX cannot. There are also other mechanisms in order to ensure synchronization to resources, and these are Semaphores, Conditional Variables or Condvars, Barriers, and Sleepons. These are not discussed here, but they provide synchronization services as a part of their duties. If you equate the principles discussed here to Cisco IOS, then Cisco IOS is a single process operating many threads, with all threads that have access to the same memory space. But, Cisco IOS calls these threads processes.

Process and Thread States

Within Cisco IOS XR there are servers who provide the services and clients who use the services. A particular process can have a number of threads who provide the same service. Another process can have a number of clients that might require a particular service at any point in time. Access to the servers is not always available, and if a client request access to a service it sits there and waits for the server to be free. In this case the client is said to be blocked. This is called a blocking client server model. The client might be blocked because it waits for a resource such as a MUTEX, or due to the fact that the server has not yet replied.

Issue a **show process ospf** command in order to check the status of the threads in the ospf process:

```
RP/0/RP1/CPU0:CWDCRS#show process ospf
                  Job Id: 250
                     PID: 110795
         Executable path: /disk0/hfr−rout−3.2.3/bin/ospf
              Instance #: 1
              Version ID: 00.00.0000
                 Respawn: ON
           Respawn count: 1
  Max. spawns per minute: 12
            Last started: Tue Jul 18 13:10:06 2006
           Process state: Run
           Package state: Normal
       Started on config: cfg/gl/ipv4−ospf/proc/101/ord_a/routerid
                    core: TEXT SHAREDMEM MAINMEM
               Max. core: 0
               Placement: ON
            startup_path: /pkg/startup/ospf.startup
                   Ready: 1.591s
               Available: 5.595s
        Process cpu time: 89.051 user, 0.254 kernel, 89.305 total
JID    TID  Stack pri state        HR:MM:SS:MSEC NAME
250    1      40K   10 Receive       0:00:11:0509 ospf
250    2      40K   10 Receive       0:01:08:0937 ospf
250    3      40K   10 Receive       0:00:03:0380 ospf
250    4      40K   10 Condvar       0:00:00:0003 ospf
250    5      40K   10 Receive       0:00:05:0222 ospf
```

Note that ospf process is given a Job ID (JID), which is 250. This never changes on a running router and generally on a particular version of Cisco IOS XR. Within the ospf process there are five threads each with their own Thread ID (TID). Listed is the stack space for each thread, the priority of each thread and its state.

Synchronous Message Passing

It is mentioned earlier that QNX is a message passing operating system. It is actually a synchronous message passing operating system. A lot of the operating system issues are reflected at the synchronous messaging. It is not said that synchronous message passing causes any problems, but rather the problem symptom is reflected in the synchronous message passing. Because it is synchronous, life cycle or state information is easily accessible to the CRS−1 operator, which aids in the troubleshooting process. The message passing life cycle is similar to this:

- A server creates a message channel.
- A client connects to the channel of a server (analogous to posix open).
- A client sends a message to a server (MsgSend) and waits for a reply and blocks.
- The server receives (MsgReceive) a message from a client, processes the message, and replies to the client.
- The client unblocks and processes the reply from the server.

This blocking client−server model is the synchronous message passing. This means the client sends a message and blocks. The server receives the message, processes it, replies back to the client and then the client unblocks. These are the specific details:

- Server waits in RECEIVE state.
- Client sends a message to the server and becomes BLOCKED.
- Server receives the message and unblocks, if waiting in receive state.
- Client moves to the REPLY state.
- Server moves to the RUNNING state.
- Server processes the message.
- Server replies to the client.
- Client unblocks.

Issue the **show process** command in order to see in what states the client and servers are.

```
RP/0/RP1/CPU0:CWDCRS#show processes
JID    TID  Stack pri state        HR:MM:SS:MSEC NAME
1      1       0K  0  Ready       320:04:04:0649 procnto−600−smp−cisco−instr
1      3       0K  10 Nanosleep     0:00:00:0043 procnto−600−smp−cisco−instr
1      5       0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      7       0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      8       0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      11      0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      12      0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      13      0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      14      0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      15      0K  19 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      16      0K  10 Receive       0:02:01:0207 procnto−600−smp−cisco−instr
1      17      0K  10 Receive       0:00:00:0015 procnto−600−smp−cisco−instr
1      21      0K  10 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      23      0K  10 Running       0:07:34:0799 procnto−600−smp−cisco−instr
1      26      0K  10 Receive       0:00:00:0001 procnto−600−smp−cisco−instr
1      31      0K  10 Receive       0:00:00:0001 procnto−600−smp−cisco−instr
1      33      0K  10 Receive       0:00:00:0000 procnto−600−smp−cisco−instr
1      39      0K  10 Receive       0:13:36:0166 procnto−600−smp−cisco−instr
1      46      0K  10 Receive       0:06:32:0015 procnto−600−smp−cisco−instr
1      47      0K  56 Receive       0:00:00:0029 procnto−600−smp−cisco−instr
1      48      0K  10 Receive       0:00:00:0001 procnto−600−smp−cisco−instr
1      72      0K  10 Receive       0:00:00:0691 procnto−600−smp−cisco−instr
1      73      0K  10 Receive       0:00:00:0016 procnto−600−smp−cisco−instr
1      78      0K  10 Receive       0:09:18:0334 procnto−600−smp−cisco−instr
1      91      0K  10 Receive       0:09:42:0972 procnto−600−smp−cisco−instr
1      95      0K  10 Receive       0:00:00:0011 procnto−600−smp−cisco−instr
1      103     0K  10 Receive       0:00:00:0008 procnto−600−smp−cisco−instr
74     1       8K  63 Nanosleep     0:00:00:0001 wd−mbi
53     1      28K  10 Receive       0:00:08:0904 dllmgr
53     2      28K  10 Nanosleep     0:00:00:0155 dllmgr
53     3      28K  10 Receive       0:00:03:0026 dllmgr
53     4      28K  10 Receive       0:00:09:0066 dllmgr
53     5      28K  10 Receive       0:00:01:0199 dllmgr
270    1      36K  10 Receive       0:00:36:0091 qsm
270    2      36K  10 Receive       0:00:13:0533 qsm
270    5      36K  10 Receive       0:01:01:0619 qsm
270    7      36K  10 Nanosleep     0:00:22:0439 qsm
270    8      36K  10 Receive       0:00:32:0577 qsm
67     1      52K  19 Receive       0:00:35:0047 pkgfs
67     2      52K  10 Sigwaitinfo   0:00:00:0000 pkgfs
67     3      52K  19 Receive       0:00:30:0526 pkgfs
67     4      52K  10 Receive       0:00:30:0161 pkgfs
67     5      52K  10 Receive       0:00:25:0976 pkgfs
68     1       8K  10 Receive       0:00:00:0003 devc−pty
52     1      40K  16 Receive       0:00:00:0844 devc−conaux
52     2      40K  16 Sigwaitinfo   0:00:00:0000 devc−conaux
```

```
52     3      40K  16 Receive       0:00:02:0981 devc−conaux
52     4      40K  16 Sigwaitinfo   0:00:00:0000 devc−conaux
52     5      40K  21 Receive       0:00:03:0159 devc−conaux
65545  2      24K  10 Receive       0:00:00:0487 pkgfs
65546  1      12K  16 Reply         0:00:00:0008 ksh
66     1       8K  10 Sigwaitinfo   0:00:00:0005 pipe
66     3       8K  10 Receive       0:00:00:0000 pipe
66     4       8K  16 Receive       0:00:00:0059 pipe
66     5       8K  10 Receive       0:00:00:0149 pipe
66     6       8K  10 Receive       0:00:00:0136 pipe
71     1      16K  10 Receive       0:00:09:0250 shmwin_svr
71     2      16K  10 Receive       0:00:09:0940 shmwin_svr
61     1       8K  10 Receive       0:00:00:0006 mqueue
```

Blocked Process and Process States

Issue the **show process blocked** command in order to see what process are in blocked state.

```
RP/0/RP1/CPU0:CWDCRS#show processes blocked
  Jid       Pid Tid                 Name State  Blocked−on
65546      4106   1                  ksh Reply    4104  devc−conaux
  105     61495   2              attachd Reply   24597  eth_server
  105     61495   3              attachd Reply    8205  mqueue
  316     65606   1          tftp_server Reply    8205  mqueue
  233     90269   2              lpts_fm Reply   90223  lpts_pa
  325    110790   1            udp_snmpd Reply   90257  udp
  253    110797   4               ospfv3 Reply   90254  raw_ip
  337    245977   2               fdiagd Reply   24597  eth_server
  337    245977   3               fdiagd Reply    8205  mqueue
65762   5996770   1                 exec Reply       1  kernel
65774   6029550   1                 more Reply    8203  pipe
65778   6029554   1       show_processes Reply       1  kernel
RP/0/RP1/CPU0:CWDCRS#
```

Synchronized message passing enables you to easily track the life cycle of inter−process communication between the different threads. At any time, a thread can be in a specific state. A blocked state can be a symptom of a problem. This does not mean that if a thread is in blocked state then there is a problem, so do not issue the **show process blocked** command and open a case with Cisco Technical Support. Blocked threads are also very normal.

Note the previous output. If you look at the first thread in the list, note it is the ksh, and its reply is blocked on devc−conaux. The client, the ksh in this case, sent a message to the devc−conaux process, the server, which is devc−conaux, holds ksh reply blocked until it replies. Ksh is the UNIX shell that someone uses on the console or AUX port. Ksh waits for input from the console, and if there is none because the operator is not typing, then it remains blocked until such time that it processes some input. After the processing, ksh returns to reply blocked on devc−conaux.

This is normal and does not illustrate a problem. The point is that blocked threads are normal, and it depends on what XR version, the type of system you have, what you have configured and who does what that alters the output of the **show process blocked** command. The use of the **show process blocked** command is a good way to start to troubleshoot OS type problems. If there is a problem, for instance CPU is high, then use the previous command in order to see if anything looks outside of normal.

Understand what is normal for your functioning router. This provides a baseline for you to use as a comparison when you troubleshoot process life cycles.

At any time, a thread can be in a particular state. This table provides a list of the states:

If the State is:	The Thread is:
DEAD	Dead. The Kernel is waiting to release the threads resources.
RUNNING	Actively running on a CPU
READY	Not running on a CPU but is ready to run
STOPPED	Suspended (SIGSTOP signal)
SEND	Waiting for a server to receive a message
RECEIVE	Waiting for a client to send a message
REPLY	Waiting for a server to reply to a message
STACK	Waiting for more stack to be allocate
WAITPAGE	Waiting for the process manager to resolve a page fault
SIGSUSPEND	Waiting for a signal
SIGWAITINFO	Waiting for a signal
NANOSLEEP	Sleeping for a period of time
MUTEX	Waiting to acquire a MUTEX
CONDVAR	Waiting for a conditional variable to be signaled
JOIN	Waiting for the completion of another thread
INTR	Waiting for an interrupt
SEM	Waiting to acquire a semaphore

Important Processes and Their Functions

Cisco IOS XR has many processes. These are some important ones with their functions explained here.

WatchDog System Monitor (WDSysmon)

This is a service provided for the detection of process hangs and low memory conditions. Low memory can occur as a result of a memory leak or some other extraneous circumstance. A hang can be the result of a number of conditions such as process deadlocks, infinite loops, kernel lockups or scheduling errors. In any multi−threaded environment the system can get in a state known as a deadlock condition, or just simply deadlock. A deadlock can occur when one or more threads are unable to continue due to resource contention. For example, thread A can send a message to thread B while simultaneously thread B sends a message to thread A. Both threads wait on each other and can be in send blocked state, and both threads wait forever. This is a simple case that involves two threads, but if a server is responsible for a resource that is used by many threads is blocked on another thread, then the many threads that request access to that resource can be send blocked waiting on the server.

Deadlocks can occur between a few threads, but can impact other threads as a result. Deadlocks are avoided by good program design, but irrespective of how magnificently a program is designed and written. Sometimes a particular sequence of events that are data dependent with specific timings can cause a deadlock. Deadlocks are not always deterministic and generally are very difficult to reproduce. WDSysmon has many threads with one that runs at the highest priority that Neutrino supports, 63. Running at priority 63 ensures it that thread gets CPU time in a priority based preemptive scheduling environment. WDSysmon works with the hardware watchdog capability and watches over the software processes that look for hang conditions. When such conditions are detected, WDSysmon collects further information around the condition, can coredump the process or the kernel, write out to syslogs, run scripts, and kill the deadlocked processes. Dependent upon how drastic the problem is, it can initiate a Route Processor switch over in order to maintain system operation.

```
RP/0/RP1/CPU0:CWDCRS#show processes wdsysmon
                  Job Id: 331
                     PID: 36908
         Executable path: /disk0/hfr-base-3.2.3/sbin/wdsysmon
              Instance #: 1
              Version ID: 00.00.0000
                 Respawn: ON
           Respawn count: 1
  Max. spawns per minute: 12
            Last started: Tue Jul 18 13:07:36 2006
           Process state: Run
           Package state: Normal
                    core: SPARSE
               Max. core: 0
                   Level: 40
               Mandatory: ON
            startup_path: /pkg/startup/wdsysmon.startup
            memory limit: 10240
                   Ready: 0.705s
        Process cpu time: 4988.295 user, 991.503 kernel, 5979.798 total

JID    TID  Stack pri state        HR:MM:SS:MSEC NAME
331    1      84K  19 Receive       0:00:00:0029 wdsysmon
331    2      84K  10 Receive       0:17:34:0212 wdsysmon
331    3      84K  10 Receive       0:00:00:0110 wdsysmon
331    4      84K  10 Receive       1:05:26:0803 wdsysmon
331    5      84K  19 Receive       0:00:06:0722 wdsysmon
331    6      84K  10 Receive       0:00:00:0110 wdsysmon
331    7      84K  63 Receive       0:00:00:0002 wdsysmon
331    8      84K  11 Receive       0:00:00:0305 wdsysmon
331    9      84K  20 Sem           0:00:00:0000 wdsysmon
```

The process WDSysmon has nine threads. Four run at priority 10, the other four are at 11, 19, 20 and 63. When a process is designed, the programmer carefully considers the priority that each thread within the process should be given. As discussed previously, the scheduler is priority based, which means a higher priority thread always preempts one of lower priority. Priority 63 is the highest priority a thread can run at, which is thread 7 in this case. Thread 7 is the watcher thread, the thread that tracks CPU hogs. It must run at a higher priority than the other threads that it watches otherwise it might not get the chance to run at all, which prevents it from the steps that it was designed to perform.

Netio

In Cisco IOS, there is the concept of fast switching and process switching. Fast switching uses the CEF code and occurs at interrupt time. Process switching uses ip_input, which is the IP switching code, and is a scheduled process. On higher end platforms CEF switching is done in hardware, and ip_input is scheduled on the CPU. The equivalent of ip_input in Cisco IOS XR is Netio.

```
P/0/RP1/CPU0:CWDCRS#show processes netio
                  Job Id: 241
                     PID: 65602
         Executable path: /disk0/hfr-base-3.2.3/sbin/netio
              Instance #: 1
                    Args: d
              Version ID: 00.00.0000
                 Respawn: ON
           Respawn count: 1
```

```
  Max. spawns per minute: 12
            Last started: Tue Jul 18 13:07:53 2006
           Process state: Run
           Package state: Normal
                    core: DUMPFALLBACK COPY SPARSE
               Max. core: 0
                   Level: 56
               Mandatory: ON
            startup_path: /pkg/startup/netio.startup
                   Ready: 17.094s
        Process cpu time: 188.659 user, 5.436 kernel, 194.095 total
JID    TID  Stack pri state        HR:MM:SS:MSEC NAME
241    1     152K  10 Receive       0:00:13:0757 netio
241    2     152K  10 Receive       0:00:10:0756 netio
241    3     152K  10 Condvar       0:00:08:0094 netio
241    4     152K  10 Receive       0:00:22:0016 netio
241    5     152K  10 Receive       0:00:00:0001 netio
241    6     152K  10 Receive       0:00:04:0920 netio
241    7     152K  10 Receive       0:00:03:0507 netio
241    8     152K  10 Receive       0:00:02:0139 netio
241    9     152K  10 Receive       0:01:44:0654 netio
241    10    152K  10 Receive       0:00:00:0310 netio
241    11    152K  10 Receive       0:00:13:0241 netio
241    12    152K  10 Receive       0:00:05:0258 netio
```

Group Services Process (GSP)

There is a need for communication in any supercomputer with several thousand nodes that each run its own instance of the kernel. In the Internet, one to many communication is efficiently done via multicasting protocols. GSP is the internal multicasting protocol that is used for IPC within CRS−1. GSP provides one to many reliable group communication that is connectionless with asynchronous semantics. This allows GSP to scale to the thousand of nodes.

```
RP/0/RP1/CPU0:CWDCRS#show processes gsp
                  Job Id: 171
                     PID: 65604
         Executable path: /disk0/hfr−base−3.2.3/bin/gsp
              Instance #: 1
              Version ID: 00.00.0000
                 Respawn: ON
           Respawn count: 1
  Max. spawns per minute: 12
            Last started: Tue Jul 18 13:07:53 2006
           Process state: Run
           Package state: Normal
                    core: TEXT SHAREDMEM MAINMEM
               Max. core: 0
                   Level: 80
               Mandatory: ON
            startup_path: /pkg/startup/gsp−rp.startup
                   Ready: 5.259s
               Available: 16.613s
        Process cpu time: 988.265 user, 0.792 kernel, 989.057 total
JID    TID  Stack pri state        HR:MM:SS:MSEC NAME
171    1     152K  30 Receive       0:00:51:0815 gsp
171    3     152K  10 Condvar       0:00:00:0025 gsp
171    4     152K  10 Receive       0:00:08:0594 gsp
171    5     152K  10 Condvar       0:01:33:0274 gsp
171    6     152K  10 Condvar       0:00:55:0051 gsp
171    7     152K  10 Receive       0:02:24:0894 gsp
171    8     152K  10 Receive       0:00:09:0561 gsp
171    9     152K  10 Condvar       0:02:33:0815 gsp
171    10    152K  10 Condvar       0:02:20:0794 gsp
171    11    152K  10 Condvar       0:02:27:0880 gsp
```

```
       171    12    152K  30 Receive       0:00:46:0276 gsp
       171    13    152K  30 Receive       0:00:45:0727 gsp
       171    14    152K  30 Receive       0:00:49:0596 gsp
       171    15    152K  30 Receive       0:00:38:0276 gsp
       171    16    152K  10 Receive       0:00:02:0774 gsp
```

BCDL Bulk Content Downloader

BCDL is used in order to reliably multicast data to various nodes such as RPs and MSCs. It uses GSP as the underlying transport. BCDL guarantees **in order** delivery of messages. Within BCDL there is an agent, a producer and a consumer. The agent is the process that communicates with the producer in order to retrieve and buffer the data before its multicasts to the consumers. The producer is the process that produces the data that everyone wants, and the consumer is the process interested to receive the data provided by the producer. BCDL is used during Cisco IOS XR software upgrades.

Lightweight Messaging (LWM)

LWM is a Cisco−created form of messaging that was designed to create a layer of abstraction between the applications that inter process communicate with each other and Neutrino, with the goal as independence of the operating system and the transport layer. If Cisco desires to change the OS vendor from QNX to someone else, a layer of abstraction between the rudimentary functions of the underlying operating system helps remove the dependency on the operating system and aids in porting to another operating system. LWM provides synchronous guaranteed message delivery, which like native Neutrino message passing, causes the sender to block until the receiver replies.

LWM also provides asynchronous message delivery via 40 bit pulses. Asynchronous messages are asynchronously sent, which means the message is queued and the sender does not block, but are not received by the server asynchronously, but when the server polls for the next available message. LWM is structured as client/server. The server creates a channel that gives it an **ear** to listen in for messages and sits in a while the loop does a message receive listening on the channel, which it just created. When a message arrives it unblocks and gets a client identifier, which is effectively the same thing as the receive ID from the message received. The server then performs some processing and later does a message reply to the client identifier.

On the client side it does a message connect. It gets passed an identifier to whom it connects and then does a message send and is blocked. When the server finishes processing, it replies and the client becomes unblocked. This is the virtually the same as Neutrinos native message passing, so the layer of abstraction is very thin.

LWM is designed with a minimum number of system calls and context switches for high performance, and is the preferred method of IPC in the Cisco IOS XR environment.

Envmon

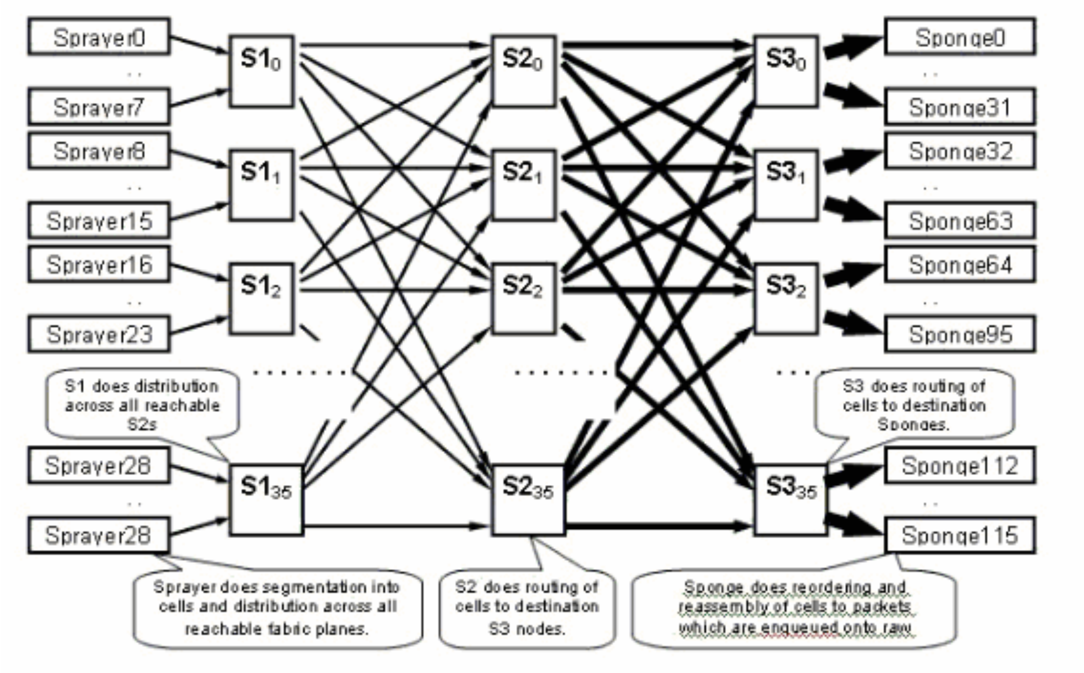
At the most fundamental level, the environmental monitor system is responsible for warning when physical parameters, for example temperature, voltage, fan speed and so forth, fall outside of operational ranges, and shutting down hardware that approaches critical levels where hardware might be damaged. It periodically monitors each available hardware sensor, compares the measured value against card−specific thresholds, and raises alarms as necessary in order to accomplish this task. A persistent process, started at system initialization, which periodically polls all hardware sensors, for example voltage, temperature, and fan speed, in the chassis and provides this data to external management clients. In addition, the periodic process compares sensor readings with alarm thresholds and publishes environmental alerts to System database for subsequent action by the Fault Manager. If the sensor readings are dangerously out of range, the environmental monitoring process might cause the card to be shutdown.

CRS−1 Fabric Introduction

- Multistage Fabric£ Stage Benes topology
- Dynamic routing within the fabric to minimize congestion
- Cell based: 136 byte cells, 120 byte data payload
- Flow control to improve traffic isolation and minimize buffering requirements in the fabric
- Stage to Stage Speedup delivering
- Two Casts of traffic supported (Unicast & Multicast)
- Two Priorities of traffic supported per cast (high and low)
- Support for 1M Fabric multicast groups (FGIDs)
- Cost effective fault tolerance : N+1 or N+k redundancy using fabric planes as opposed to 1+1 at greatly increased cost

When you run in single−chassis mode, the S1, S2 and S3 asics are located on the same fabric cards. This card is also commonly referred to as **S123 card**. In a Multi−Chassis configuration, the S2 is separated and it is on the Fabric Card Chassis (FCC). This configuration requires two fabric cards to form a plane, an S2 card, and a S13 card. Each MSC connects to eight fabric planes in order to provide redundancy so that if you loose one or more planes, your fabric still passes traffic although the aggregate traffic, which can go through the fabric, is lower. The CRS can still operate at linerate for most packet sizes with only seven planes. Backpressure is sent over the fabric over an odd and even plane. It is not recommended to run a system with less than two planes, in an odd and even plane. Anything less than two planes is not a supported configuration.

The Fabric Plane


Sprayer0
Sprayer7
Sprayer8
Sprayer15
Sprayer16
Sprayer23
Sprayer28
Sprayer28
$S1_0$
$S1_1$
$S1_2$
$S1_{35}$
$S2_0$
$S2_1$
$S2_2$
$S2_{35}$
$S3_0$
$S3_1$
$S3_2$
$S3_{35}$
Sponge0
Sponge31
Sponge32
Sponge63
Sponge64
Sponge95
Sponge112
Sponge115
S1 does distribution across all reachable S2s
S3 does routing of cells to destination Sponges.
Sprayer does segmentation into cells and distribution across all reachable fabric planes.
S2 does routing of cells to destination S3 nodes.
Sponge does reordering and reassembly of cells to packets which are enqueued onto raw

The previous diagram represents one plane. You have to multiply that diagram by eight. That means that the sprayer (ingressq) asic of a LC connects to 8 S1s (1 S1 per plane). The S1 in each fabric plane connects to 8 sprayers:

- the 8 top LCs of the chassis
- the 8 bottom LCs

There are 16 S1s per 16 slot LC chassis: 8 for the top LCs (1 per plane) + 8 for the bottom LCs.

On a single 16 slot chassis, an S123 fabric card has 2 S1s, 2 S2 and 4 S3s. That is part of the fabric speedup computation. There is twice as much traffic, which can exit the fabric as traffic can enter. There are also currently two sponges (fabricq) per LC compared to 1 Sprayer. This allows for buffering on the egress LC when more than one ingress LCs overload an egress LC. The egress LC is able to absorb that extra bandwidth from the fabric.

Fabric Monitoring

Plane availability and connectivity:

```
admin show controller fabric plane all
admin show controller fabric connectivity all detail
```

Check if planes are receiving/transmitting cells and some errors are incrementing:

```
admin show controllers fabric plane all statistics
```

The acronyms in the previous command:

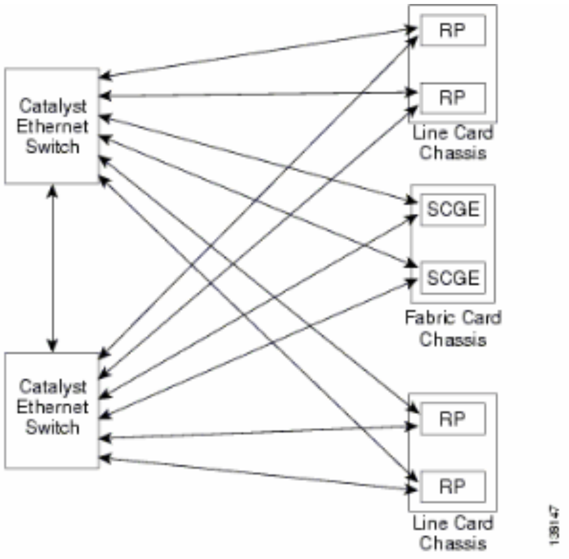
- CE Correctable Error
- UCE Uncorrectable Error
- PE Parity Error

Do not worry if they notice a few errors, as this can happen on bootup. The fields should not be incrementing at run time. If they are, it can be an indication of a problem in the fabric. Issue this command in order to get a breakdown of the errors per fabric plane:

```
admin show controllers fabric plane <0−7> statistics detail
```

Control Plane overview

The control plane connectivity between the line−card chassis and fabric chassis is currently via Gigabit Ethernet ports on the RPs (LCC) and SCGE (FCC). The interconnection between the ports are provided via a pair of Catalyst 6500 switches, which can be connected via two or more gigabit Ethernet ports.



Catalyst 6500 Configuration

This is recommended configuration for Catalyst switches used for the multi−chassis control plane:

- A single VLAN is used on all ports.
- All ports run in access mode (no trunking).
- Spanning−tree 802.1w/s is used for loop prevention.
- Two or more links are used in order to cross−connect the two switches and STP is used for loop−prevent. Channelling is not recommended.
- Ports that connect to CRS−1 RP and SCGE use pre−standard mode since IOS−XR does not support the standards based 802.1s.
- UDLD should be enabled on the ports that connect between the switches and between the switches and the RP/SCGE.
- UDLD is enabled by default on the CRS−1.

Refer to Bringing Up the Cisco IOS XR Software on a Multishelf System for more information on how to configure a Catalyst 6500 in a Multishelf system.

Multi−Chassis Control Plane Management

The Catalyst 6504−E chassis, which provides the control plane connectivity for the multi−chassis system, is configured for these management services:

- In−band management via port gigabit 1/2, which connects to a LAN switch at each PoP. Access is only permitted for a small range of subnets and protocols.
- NTP is used in order to set the system time.
- Syslogging is performed to the standard hosts.
- SNMP polling and traps can be enabled for critical functions.

Note: No changes should be made to the Catalyst in operation. Prior testing should be done on any planned change and it is highly recommended that this is done during a maintenance window.

This is a sample of management configuration:

```
#In−band management connectivity
interface GigabitEthernet2/1
 description *CRS Multi−chassis Management Ethernet − DO NOT TOUCH*
 ip address [ip address] [netmask]
 ip access−group control_only in
!
!
ip access−list extended control_only
 permit udp [ip address] [netmask] any eq snmp
 permit udp [ip address] [netmask] eq ntp any
 permit tcp [ip address] [netmask] any eq telnet

#NTP

ntp update−calendar
ntp server [ip address]

#Syslog
logging source−interface Loopback0
logging [ip address]
logging buffered 4096000 debugging
no logging console

#RADIUS
```

```
aaa new−model
aaa authentication login default radius enable
enable password {password}
radius−server host [ip address] auth−port 1645 acct−port 1646
radius−server key {key}

#Telnet and console access
!
access−list 3 permit [ip address]
!
line con 0
 exec−timeout 30 0
 password {password}
line vty 0 4
 access−class 3 in
 exec−timeout 0 0
password {password}
```

ROMMON and Monlib

Cisco monlib is an executable program that is stored on the device and loaded into RAM for execution by ROMMON. ROMMON uses monlib in order to access files on the device. ROMMON versions can be upgraded and should be done so under recommendation of Cisco Technical Support. The latest ROMMON version is 1.40.

Upgrade Instructions

Complete these steps:

1. Download the ROMMON binaries from Cisco CRS−1 ROMMON (registered customers only) .
2. Unpack the TAR file and copy the 6 BIN files into the CRS root directory of Disk0.

   ```
   RP/0/RP0/Router#dir disk0:/*.bin

   Directory of disk0:

   65920       −rwx  360464      Fri Oct 28 12:58:02 2005  rommon−hfr−ppc7450−sc−dsmp−A
   66112       −rwx  360464      Fri Oct 28 12:58:03 2005  rommon−hfr−ppc7450−sc−dsmp−B
   66240       −rwx  376848      Fri Oct 28 12:58:05 2005  rommon−hfr−ppc7455−asmp−A.bi
   66368       −rwx  376848      Fri Oct 28 12:58:06 2005  rommon−hfr−ppc7455−asmp−B.bi
   66976       −rwx  253904      Fri Oct 28 12:58:08 2005  rommon−hfr−ppc8255−sp−A.bin
   67104       −rwx  253492      Fri Oct 28 12:58:08 2005  rommon−hfr−ppc8255−sp−B.bin
   ```
3. Use the **show diag | inc ROM|NODE|PLIM** command in order to see current rommon version.

   ```
   RP/0/RP0/CPU0:ROUTER(admin)#show diag | inc ROM|NODE|PLIM
   NODE 0/0/SP : MSC(SP)
     ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
   PLIM 0/0/CPU0 : 4OC192−POS/DPT
     ROMMON: Version 1.19b(20050216:033559) [CRS−1 ROMMON]
   NODE 0/2/SP : MSC(SP)
     ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
   PLIM 0/2/CPU0 : 8−10GbE
     ROMMON: Version 1.19b(20050216:033559) [CRS−1 ROMMON]
   NODE 0/4/SP : Unknown Card Type
   NODE 0/6/SP : MSC(SP)
     ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
   PLIM 0/6/CPU0 : 16OC48−POS/DPT
     ROMMON: Version 1.19b(20050216:033559) [CRS−1 ROMMON]
   NODE 0/RP0/CPU0 : RP
     ROMMON: Version 1.19b(20050216:033559) [CRS−1 ROMMON]
   NODE 0/RP1/CPU0 : RP
     ROMMON: Version 1.19b(20050216:033559) [CRS−1 ROMMON]
   ```

```
NODE 0/SM0/SP : FC/S
  ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
NODE 0/SM1/SP : FC/S
  ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
NODE 0/SM2/SP : FC/S
  ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
NODE 0/SM3/SP : FC/S
ROMMON: Version 1.19b(20050216:033352) [CRS−1 ROMMON]
```

4. Go into the ADMIN mode and use the **upgrade rommon a all disk0** command in order to upgrade the ROMMON.

```
RP/0/RP0/CPU0:ROUTER#admin
RP/0/RP0/CPU0:ROUTER(admin)#upgrade rommon a all disk0
Please do not power cycle, reload the router or reset any nodes until
 all upgrades are completed.
Please check the syslog to make sure that all nodes are upgraded successfully.
If you need to perform multiple upgrades, please wait for current upgrade
 to be completed before proceeding to another upgrade.
Failure to do so may render the cards under upgrade to be unusable.
```

5. Exit ADMIN Mode and enter **show log | inc "OK, ROMMON A"** and make sure all nodes successfully upgraded. If any of the nodes fail, go back to step 4 and reprogram.

```
RP/0/RP0/CPU0:ROUTER#show logging | inc "OK, ROMMON A"
RP/0/RP0/CPU0:Oct 28 14:40:57.223 PST8: upgrade_daemon[380][360]: OK,
ROMMON A is programmed successfully.
SP/0/0/SP:Oct 28 14:40:58.249 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
SP/0/2/SP:Oct 28 14:40:58.251 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
LC/0/6/CPU0:Oct 28 14:40:58.336 PST8: upgrade_daemon[244][233]: OK,
ROMMON A is programmed successfully.
LC/0/2/CPU0:Oct 28 14:40:58.365 PST8: upgrade_daemon[244][233]: OK,
ROMMON A is programmed successfully.
SP/0/SM0/SP:Oct 28 14:40:58.439 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
SP/0/SM1/SP:Oct 28 14:40:58.524 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
LC/0/0/CPU0:Oct 28 14:40:58.530 PST8: upgrade_daemon[244][233]: OK,
ROMMON A is programmed successfully.
RP/0/RP1/CPU0:Oct 28 14:40:58.593 PST8: upgrade_daemon[380][360]: OK,
ROMMON A is programmed successfully.
SP/0/6/SP:Oct 28 14:40:58.822 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
SP/0/SM2/SP:Oct 28 14:40:58.890 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
SP/0/SM3/SP:Oct 28 14:40:59.519 PST8: upgrade_daemon[125][121]: OK,
ROMMON A is programmed successfully.
```

6. Go into the ADMIN mode and use the **upgrade rommon b all disk0** command in order to upgrade the ROMMON.

```
RP/0/RP0/CPU0:ROUTER#admin
RP/0/RP0/CPU0:ROUTER(admin)#upgrade rommon b all disk0
Please do not power cycle, reload the router or reset any nodes until
 all upgrades are completed.
Please check the syslog to make sure that all nodes are upgraded successfully.
If you need to perform multiple upgrades, please wait for current upgrade
 to be completed before proceeding to another upgrade.
Failure to do so may render the cards under upgrade to be unusable.
```

7. Exit ADMIN Mode and enter **show log | inc "OK, ROMMON B"** and make sure all nodes successfully upgraded. If any of the nodes fail, go back to step 4 and reprogram.

```
RP/0/RP0/CPU0:Router#show logging | inc "OK, ROMMON B"
RP/0/RP0/CPU0:Oct 28 13:27:00.783 PST8: upgrade_daemon[380][360]: OK,
ROMMON B is programmed successfully.
```

```
LC/0/6/CPU0:Oct 28 13:27:01.720 PST8: upgrade_daemon[244][233]: OK,
ROMMON B is programmed successfully.
SP/0/2/SP:Oct 28 13:27:01.755 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
LC/0/2/CPU0:Oct 28 13:27:01.775 PST8: upgrade_daemon[244][233]: OK,
ROMMON B is programmed successfully.
SP/0/0/SP:Oct 28 13:27:01.792 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
SP/0/SM0/SP:Oct 28 13:27:01.955 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
LC/0/0/CPU0:Oct 28 13:27:01.975 PST8: upgrade_daemon[244][233]: OK,
ROMMON B is programmed successfully.
SP/0/6/SP:Oct 28 13:27:01.989 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
SP/0/SM1/SP:Oct 28 13:27:02.087 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
RP/0/RP1/CPU0:Oct 28 13:27:02.106 PST8: upgrade_daemon[380][360]: OK,
ROMMON B is programmed successfully.
SP/0/SM3/SP:Oct 28 13:27:02.695 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
SP/0/SM2/SP:Oct 28 13:27:02.821 PST8: upgrade_daemon[125][121]: OK,
ROMMON B is programmed successfully.
```

8. The **upgrade** command just burns a special reserved section of bootflash with the new ROMMON. But the new ROMMON remains inactive until the card is reloaded. So when you the reload the card, the new ROMMON is active. Reset each node one at a time or just reset the entire router in order to do this.

```
Reload Router:
RP/0/RP0/CPU0:ROUTER#hw-module node 0/RP0/CPU0 or 0/RP1/CPU0 reload (depends on whic
RP/0/RP0/CPU0:ROUTER#reload

!--- Issue right after the first command.

Updating Commit Database.  Please wait...[OK]
Proceed with reload? [confirm]


!--- Reload each Node.  For Fan Controllers (FCx),
!--- Alarm Modules (AMx), Fabric Cards (SMx), and RPs (RPx),
!--- you must wait until the reloaded node is fully reloaded
!--- before you reset the next node of the pair.  But non-pairs
!--- can be reloaded without waiting.

RP/0/RP0/CPU0:ROUTER#hw-module node 0/RP0/CPU0 or 0/RP1/CPU0 reload

!--- This depends on which on is in Standby Mode.

RP/0/RP0/CPU0:ROUTER#hw-module node 0/FC0/SP
RP/0/RP0/CPU0:ROUTER#hw-module node 0/AM0/SP
RP/0/RP0/CPU0:ROUTER#hw-module node 0/SM0/SP

!--- Do not reset the MSC and Fabric Cards at the same time.

RP/0/RP0/CPU0:ROUTER#hw-module node 0/0/CPU
```

9. Use the **show diag | inc ROM|NODE|PLIM** command in order to check the current ROMMON version.

```
RP/0/RP1/CPU0:CRS-B(admin)#show diag | inc ROM|NODE|PLIM
NODE 0/0/SP : MSC(SP)
  ROMMON: Version 1.32(20050525:193402)  [CRS-1 ROMMON]
PLIM 0/0/CPU0 : 4OC192-POS/DPT
  ROMMON: Version 1.32(20050525:193559)  [CRS-1 ROMMON]
NODE 0/2/SP : MSC(SP)
  ROMMON: Version 1.32(20050525:193402)  [CRS-1 ROMMON]
PLIM 0/2/CPU0 : 8-10GbE
```

```
  ROMMON: Version 1.32(20050525:193559)  [CRS−1 ROMMON]
NODE 0/6/SP : MSC(SP)
  ROMMON: Version 1.32(20050525:193402)  [CRS−1 ROMMON]
PLIM 0/6/CPU0 : 16OC48−POS/DPT
  ROMMON: Version 1.32(20050525:193559)  [CRS−1 ROMMON]
NODE 0/RP0/CPU0 : RP
  ROMMON: Version 1.32(20050525:193559)  [CRS−1 ROMMON]
NODE 0/RP1/CPU0 : RP
  ROMMON: Version 1.32(20050525:193559)  [CRS−1 ROMMON]]
NODE 0/SM0/SP : FC/S
  ROMMON: Version 1.32(20050525:193402)  [CRS−1 ROMMON]
NODE 0/SM1/SP : FC/S
  ROMMON: Version 1.32(20050525:193402)  [CRS−1 ROMMON]
NODE 0/SM2/SP : FC/S
  ROMMON: Version 1.32(20050525:193402)  [CRS−1 ROMMON]
NODE 0/SM3/SP : FC/S
  ROMMON: Version 1.32(20050525:193402)  [CRS−1 ROMMON]
```

Note: On CRS−8 and Fabric Chassis, ROMMON also sets the fan speeds to the default speed of 4000 RPM.

PLIM and MSC Overview

This represents the packet flow on the CRS−1 router, and these terms are used interchangeably:

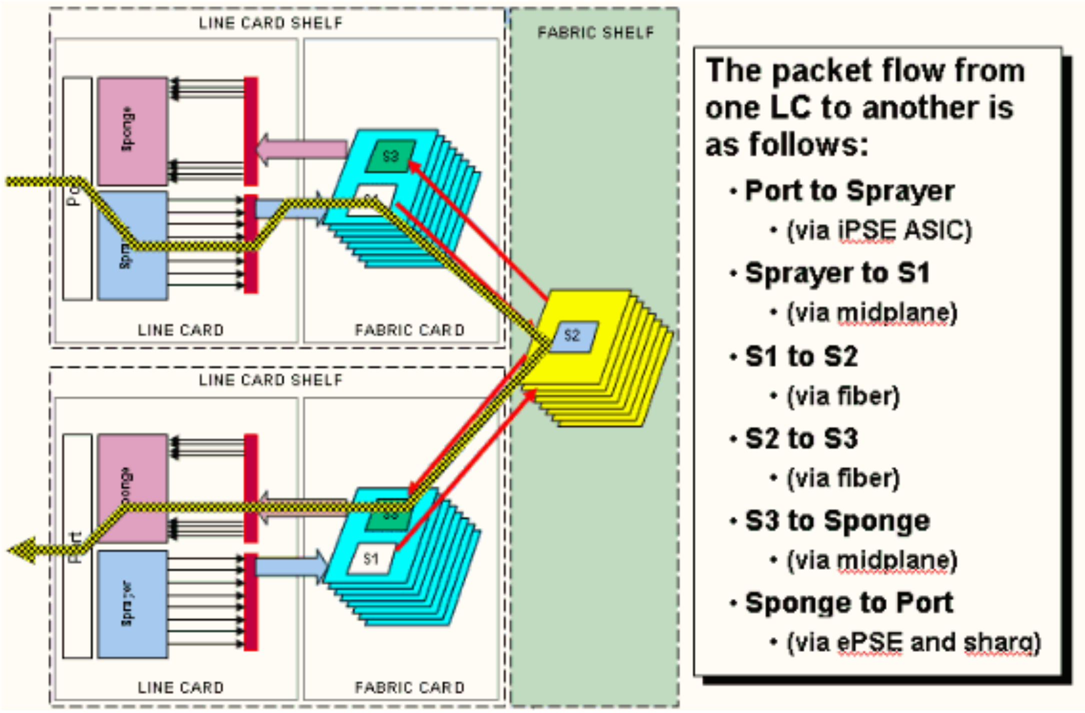
IngressQ ASIC is also called the Sprayer ASIC.

FabricQ ASIC is also called the Sponge ASIC.

EgressQ ASIC is also called the Sharq ASIC.

SPP is also called the PSE (Packet Switch Engine) ASIC.

Rx PLIM > Rx SPP > Ingress Q > Fabric > Fabric Q > Tx SPP > Egress Q > Tx PLIM (Sprayer) (Sponge) (Sharq)



The packet flow from one LC to another is as follows:

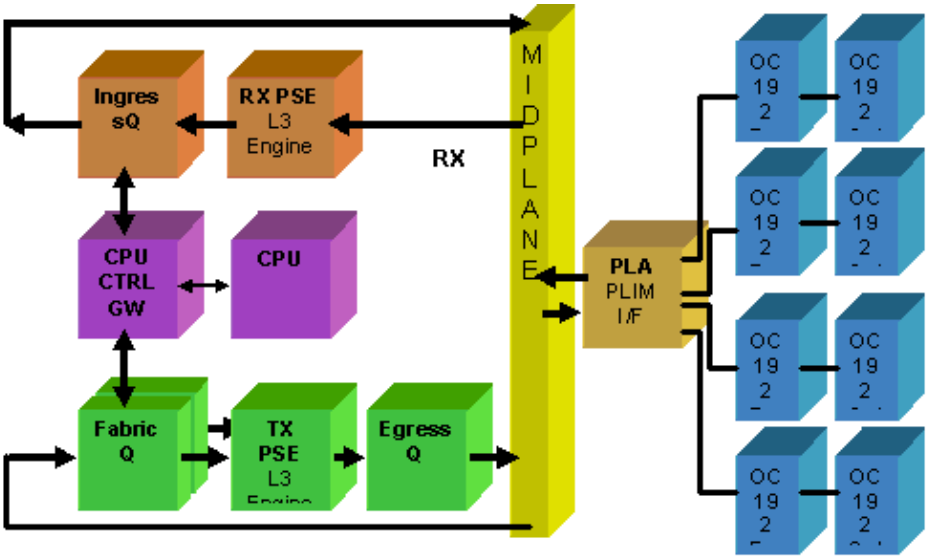
- Port to Sprayer
  - (via iPSE ASIC)
- Sprayer to S1
  - (via midplane)
- S1 to S2
  - (via fiber)
- S2 to S3
  - (via fiber)
- S3 to Sponge
  - (via midplane)
- Sponge to Port
  - (via ePSE and sharq)

Packets are received on the Physical Layer Interface Module (PLIM).

The PLIM contains the physical interfaces for the MSC with which it mates. The PLIM and MSC are separate cards connected via the chassis backplane. As a result the interface types for a particular MSC are defined by the type of the PLIM that it mated with. Dependent upon the type of PLIM, the card contains a various number of ASICs that provide the physical media and framing for the interfaces. The purpose of the PLIM ASICs are to provide the interface between the MSC and the physical connections. It terminates the fibre, does the light to electrical conversion, terminates the media framing being SDH/Sonet/Ethernet/HDLC/PPP, checks the CRC, adds some control information called the Buffer Header and forwards the bits that remain onto the MSC. The PLIM does not source/sink the HDLC or PPP keepalives. These are handled by CPU on the MSC.

The PLIM also provides these functions:

- MAC filtering for 1/10 Gigabit Ethernet
- Ingress/Egress MAC accounting for 1/10 Gigabit Ethernet
- VLAN filtering for 1/10 Gigabit Ethernet
- VLAN accounting for 1/10 Gigabit Ethernet
- Ingress buffering and congestion notification



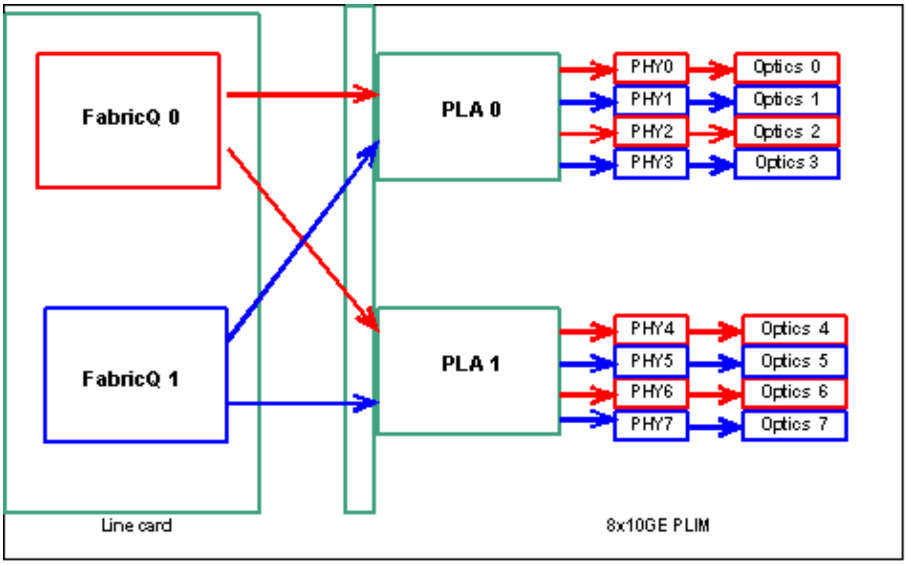
PLIM Oversubscription

10GE PLIM

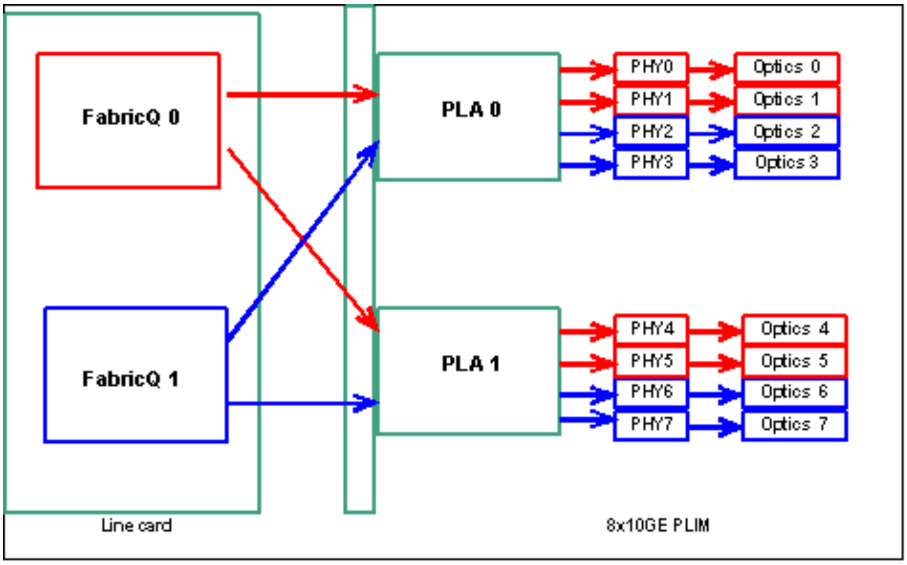
The 8 X 10G PLIM offers the capability to terminate approximately 80 Gbps of traffic whilst the forwarding capacity of the MSC is a maximum of 40 Gbps. If all of the ports available on the PLIM are populated, then oversubscription takes place and QoS modeling becomes extremely important to ensure that premium traffic is not inadvertently dropped. For some, oversubscription is not an option and must be avoided. Only four of the eight ports must be used in order to do this. In addition, care must be taken to ensure that the optimum bandwidth within the MSC and PLIM is available to each of the four ports.

Note: The port mapping changes from release 3.2.2 onwards. See these diagrams.

Port mapping up to release 3.2.1



Port mapping from release 3.2.2 onwards



As previously mentioned, the physical ports are serviced by one of the two FabricQ ASICs. The assignment of ports to the ASIC is statically defined and cannot be altered. In addition, the 8 X 10G PLIM has two PLA ASICs. The first PLA services ports 0 to 3, the second services 4 to 7. The bandwidth capacity of a single PLA on the 8 X 10G PLIM is approximately 24 Gbps. The switching capacity of a single FabricQ ASIC is approximately 62 Mpps.

If you populate port 0 to 3 or ports 4 to 7, the bandwidth capacity of the PLA (24 Gbps) are shared between all four of the ports that restrict the overall throughput. If you populate ports 0,2,4 & 6 (up to 3.2.1) or 0,1,4 & 5 (3.2.2 onwards) as all of these ports are serviced by the one FabricQ ASIC, whose switching capacity is 62 Mpps, again, which restricts the throughput capacity.

It is best to utilize the ports in a manner that obtains the highest efficiency of both the PLAs and the FabricQ ASICs in order to achieve optimum performance.

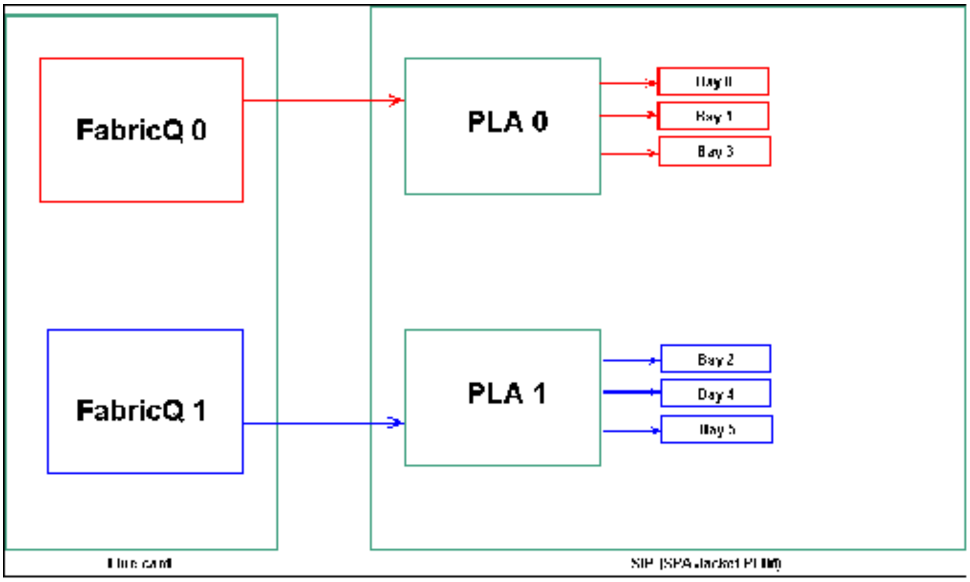
SIP−800/SPA

The SIP−800 PLIM offers the ability to operate with modular interface cards known as Service Port Adapters (SPAs). The SIP−800 provides 6 SPA bays with a theoretical interface capacity of 60 Gbps. The forwarding capacity of the MSC is a maximum of 40 Gbps. If all of the bays on the SIP−800 were to be populated, then, dependent upon the SPA type, it is possible that oversubscription takes place and QoS modeling becomes extremely important in order to ensure that premium traffic is not inadvertently dropped.

Note: Oversubscription is not supported with POS interfaces. But, placement of the 10 Gb POS SPA must be appropriate in order to ensure the correct throughput capacity is provided. The 10 Gb Ethernet SPA is only supported in IOS−XR release 3.4. This SPA offers oversubscription capabilities.

For some, oversubscription is not an option and must be avoided. Only four of the sixbays must be used in order to do this. In addition, care must be taken in order to ensure that the optimum bandwidth within the MSC and PLIM is available to each of the four ports.

SPA Bay Mapping



As mentioned in previously, the physical ports are serviced by one of the two FabricQ ASICs. The assignment of ports to the ASIC is statically defined and cannot be altered. In addition, the SIP−800 PLIM has two PLA ASICs. The first PLA services ports 0,1 & 3, the second services 2, 4 & 5.

The bandwidth capacity of a single PLA on the SIP−800 PLIM is approximately 24 Gbps. The switching capacity of a single FabricQ ASIC is approximately 62 Mpps.

If you populate port 0,1 & 3 or ports 2, 4 & 5, the bandwidth capacity of the PLA (24 Gbps) are shared between all three of the ports that restrict the overall throughput. Since a single FabricQ each services those port groups, the maximum packet rate of the port group is 62 Mpps. It is best to utilize the ports in a manner that obtains the highest efficiency of the PLAs in order to achieve optimum bandwidth.

Suggested Placement:

	SPA bay#	SPA bay#	SPA bay#	SPA bay#
Option 1	0	1	4	5
Option 2	1	2	3	4

If you want to populate the card with more than four SPA, the recommendation is to complete one of the options previously listed, which spread the interfaces between the two port groups (0,1 & 3 & 2,4 & 5). You should then place the next SPA modules in one of the open ports in either the 0,1 & 3 & 2,4 & 5 port groups.

DWDM XENPACKs

From release 3.2.2 onwards, DWDM XENPACKs can be installed and provide **tunable** optics modules. The cooling requirements of such XENPACK modules requires that there be a blank slot between installed modules. In addition, if a single DWDM XENPACK module is installed, a maximum of four ports can be used, even if the the XENPACK modules are not DWDM devices. This therefore has a direct impact on the FabricQ to PLA to Port mapping. Attention needs to be paid to this requirement and is considered in this table.

Suggested Placement:

	Optics port #	Optics port #	Optics port #	Optics port #
Option 1 or DWDM XENPACK	0	2	5	7
Option 2	1	3	4	6

For a 3.2.2 or later or 3.3 installation, avoid the FabricQ mapping change. A simpler placement pattern can therefore be used for both regular and DWDM XENPACK modules.

	Optics port #	Optics port #	Optics port #	Optics port #
Option 1	0	2	4	6
Option 2	1	3	5	7

If you want to populate the card with more than four non−DWDM XENPACK ports, the recommendation is to complete one of the options listed, which spreads the Optical interface modules between the two port groups (0−3 & 4−7). You need to then place the next Optical interface modules in one of the open ports in either the 0−3 or 4−7 port groups. If you use the 0−3 port group for Optical interface module #5, the Optical interface modules #6 should be placed in the 4−7 port group.

Refer to DWDM XENPAK Modules for more details.

Configuration Management

Configuration in IOS−XR is done through a two stage configuration, the configuration is entered by the user in the first stage. This is the stage where only configuration syntax is checked by the CLI. The configuration entered in this stage is only known to the configuration agent process, for example, CLI/XML. The configuration is not verified since it is not written to the sysdb server. The backend application are not notified and cannot access or have any knowledge of the configuration in this stage.

In the second stage, the configuration is explicitly committed by the user. In this stage the configuration is written to the sysdb server, backend applications verify the configurations and notifications are generated by sysdb. You can abort a configuration session before you commit the configuration entered in the first stage. So, it is not safe to assume that all configuration entered in the stage one is always committed in stage two.

In addition, the operation and/or running configuration of the router can be modified by multiple users during stage one and stage two. So, any test of router that runs configuration and/or operational state in stage one might not be valid in stage two where the configuration is actually committed.

Configuration File Systems

Configuration File System (CFS) are a set of files and directories used in order to store the configuration of the router. CFS is stored under the directory disk0:/config/, which is the default media used on the RP. Files and directories in CFS are internal to the router and should never be modified or removed by the user. This can result in loss or corruption of the configuration and affects service.

The CFS is checkpointed to the standby−RP after every commit. This helps preserve the configuration file of the router after a fail over.

During router bootup, the last active configuration is applied from the configuration commit database stored in CFS. It is not necessary for the user to manually save the active configuration after each configuration commit, since this is done automatically by the router.

It is not advisable to make configuration changes while configuration is being applied during bootup. If the configuration application is not complete, you see this message when you log on to the router:

System Configuration Process

The startup configuration for this device is presently loading. This can take a few minutes. You are notified upon completion. Please do not attempt to reconfigure the device until this process is complete. In some rare cases, it might be desirable to restore router configuration from a user provided ASCII configuration file instead of restoring the last active configuration from CFS.

You can force the application of a configuration file by:

```
using the  −a  option with the boot command. This option forces
    the use of the specified file only for this boot.

    rommon>boot <image>  a <config−file−path>

    setting the value of  IOX_CONFIG_FILE  boot variable to the
    path of configuration file. This forces the use of the specified file
    for all boots while this variable is set.

    rommon>IOX_CONFIG_FILE=<config−file−path>
    rommon>boot <image>
```

While you restore the router configuration, one or more configuration item(s) might fail to take effect. All failed configuration is saved in the CFS and is maintained until the next reload.

You can browse the failed configuration, address the errors and re−apply the configuration.

These are some tips in order to address failed configuration during startup of the router.

In IOX, configuration can be classified as failed configuration for three reasons:

1. Syntax errors The parser generates syntax errors, which usually indicate that there is an incompatibility with CLI commands. You should correct the syntax errors and reapply the configuration.
2. Semantic errors Semantic errors are generated by the backend components when the configuration manager restores the configuration during startup of the router. It is important to note that cfgmgr is not responsible for ensuring the configuration is accepted as part of running configuration. Cfgmgr is merely a **middle−man** and only reports any semantic failures that backend components generate. It is up to each backend component owner to analyze the failure reason and determine the reason for the failure. Users can execute the **describe <CLI commands>** from the configuration mode in order to easily find the owner of the backend component verifier. For example, if the **router bgp 217** shows up as failed configuration, the **describe** command shows that the component verifier is the ipv4−bgp component.

   ```
   RP/0/0/CPU0:router#configure terminal
   RP/0/0/CPU0:router(config)#describe router bgp 217
   The command is defined in bgpv4_cmds.parser

   Node 0/0/CPU0 has file bgpv4_cmds.parser for boot package /gsr−os−mbi−3.3.87/mbi1200
   Package:
       gsr−rout
           gsr−rout V3.3.87[Default]  Routing Package
           Vendor : Cisco Systems
           Desc   : Routing Package
           Build  : Built on Mon Apr  3 16:17:28 UTC 2006
           Source : By ena−view3 in /vws/vpr/mletchwo/cfgmgr_33_bugfix for c2.95.3−p8
           Card(s): RP, DRP, DRPSC
           Restart information:
             Default:
               parallel impacted processes restart
   Component:
       ipv4−bgp V[fwd−33/66]  IPv4 Border Gateway Protocol (BGP)

   File: bgpv4_cmds.parser


   User needs ALL of the following taskids:

           bgp (READ WRITE)

   It will take the following actions:
     Create/Set the configuration item:
            Path: gl/ip−bgp/0xd9/gbl/edm/ord_a/running
           Value: 0x1
   Enter the submode:
           bgp
   RP/0/0/CPU0:router(config)#
   ```
3. Apply errors The configuration has been successfully verified and accepted as part of running configuration but the backend component is not able to update its operational state for some reason. The configuration shows in both running configuration, since it was correctly verified, and as failed configuration because of the backend operational error. The **describe** command can again be run on the CLI that failed to be applied in order to find the component apply owner.

- Complete these steps in order to browse and reapply failed configuration during startup operators:

  For R3.2 operators can use this procedure in order to reapply failed configuration:

  - Operators can use the **show configuration failed startup** command in order to browse the failed configuration saved during router startup.
  - Operators should run the **show configuration failed startup noerror | file myfailed.cfg** command in order to save the startup failed configuration to a file.
  - Operators should go to **configuration** mode and use **load/commit** commands in order to reapply this failed configuration:

    ```
    RP/0/0/CPU0:router(config)#load myfailed.cfg
    Loading.
    197 bytes parsed in 1 sec (191)bytes/sec
    RP/0/0/CPU0:router(config)#commit
    ```
- For R3.3 images operators can use this updated procedure:

  Operators must use the **show configuration failed startup** command and the **load configuration failed startup** command in order to browse and reapply any failed configuration.

  ```
  RP/0/0/CPU0:router#show configuration failed startup
  !! CONFIGURATION FAILED DUE TO SYNTAX/AUTHORIZATION ERRORS
  telnet vrf default ipv4
  server max−servers 5 interface POS0/7/0/3 router static
  address−family ipv4 unicast
   0.0.0.0/0 172.18.189.1

  !! CONFIGURATION FAILED DUE TO SEMANTIC ERRORS
  router bgp 217 !!%
  Process did not respond to sysmgr !
  RP/0/0/CPU0:router#

  RP/0/0/CPU0:router(config)#load configuration failed startup noerror
  Loading.
  263 bytes parsed in 1 sec (259)bytes/sec
  RP/0/0/CPU0:mike3(config−bgp)#show configuration
  Building configuration...
  telnet vrf default ipv4 server max−servers 5 router static
  address−family ipv4 unicast
    0.0.0.0/0 172.18.189.1
   !
  !
  router bgp 217
  !
  end

  RP/0/0/CPU0:router(config−bgp)#commit
  ```

Kernel Dumper

By default IOS−XR writes a core dump to the harddisk should a process crash, but not if the kernel itself crashes. Note that for a multi−chassis system this functionality is currently only supported for the line−card chassis 0. The other chassis is supported in a future release of software.

It is suggested that kernel dumps for both the RPs and MSCs be enabled with the use of these configuration in both the standard and admin−mode configurations:

```
exception kernel memory kernel filepath harddisk:
```

```
exception dump−tftp−route port 0 host−address 10.0.2.1/16 destination 10.0.2.1 next−hop 10
```

Kernel Dump Configuration

This results in this occurrence for a kernel crash:

1. An RP crashes and a dump is written to the harddisk on that RP in the root directory of the disk.
2. If an MSC crashes, a dump is written to the harddisk of RP0 in the root directory of the disk.

This has no impact on RP failover times since non−stop forwarding (NSF) is configured for the routing protocols. It can take a few extra minutes for the crashed RP or line−card to become available again after it follows a crash while it writes the core.

An example of the addition of this configuration to both the standard and admin mode configuration is shown here. Note that the admin mode configuration requires DRPs to be used.

This output shows a Kernel Dump configuration example:

```
RP/0/RP0/CPU0:crs1#configure
RP/0/RP0/CPU0:crs1(config)#exception kernel memory kernel filepat$
RP/0/RP0/CPU0:crs1(config)#exception dump−tftp−route port 0 host−$
RP/0/RP0/CPU0:crs1(config)#commit
RP/0/RP0/CPU0:crs1(config)#
RP/0/RP0/CPU0:crs1#admin
RP/0/RP0/CPU0:crs1(admin)#configure
Session                      Line        User      Date                        Lock
00000201−000bb0db−00000000   snmp        hfr−owne  Wed Apr  5 10:14:44 2006
RP/0/RP0/CPU0:crs1(admin−config)#exception kernel memory kernel f$
RP/0/RP0/CPU0:crs1(admin−config)#exception dump−tftp−route port 0$
RP/0/RP0/CPU0:crs1(admin−config)#commit
RP/0/RP0/CPU0:crs1(admin−config)#
RP/0/RP0/CPU0:crs1(admin)#
```

Security

LPTS

Local Packet Transport Services (LPTS) handles locally destined packets. LPTS is made of various different components.

1. The main one is called the port arbitrator process. It listens to socket requests from different protocol processes, for example, BGP, IS−IS and keep track of all the binding information for those processes. For example, if a BGP process listens at socket number 179, the PA obtains that information from the BGP processes, and then assigns a binding to that process in a IFIB.
2. The IFIB, is another component of the LPTS process. It helps keep a directory of where a process is that is listening to a specific port binding. The IFIB is generated by the Port Arbitrator process and is kept with the port arbitrator. It then generates multiple subsets of this information.

   The first subset is the a slice of the IFIB. This slice can be associated to IPv4 protocol and so forth. Slices are then sent to appropriate flow managers, which then use the IFIB slice in order to forward the packet to the proper process.

   The second subset is a pre−IFIB, allows the LC to forward the packet to the proper process if only one process exists or to a proper flow manager.
3. Flow managers help further distribute the packets if the look up is non−trivial, for example, multiple processes for BGP. Each flow manager has a slice or multiple slices of the IFIB and properly

forwards packets to the appropriate processes associated with the slice of the IFIB.
4. If an entry is not defined for the destination port then it can either be dropped or forwarded to the flow manager. A packet is forwarded with no associated port if there is an associated policy for the port. The flow manager then helps generate a new session entry.

How is an Internal Packet Forwarded?

There are two types of flows, Layer 2 (HDLC, PPP) flows and Layer 4 ICMP/PING flows and routing flows.

1. Layer 2 HDLC/PPP These packets are identified by the protocol identifier and are sent directly to the CPU queues in the Sprayer. Layer 2 protocol packets get high priority and are then picked up by the CPU (via the Squid) and processed. Hence keepalives for Layer 2 are directly responded to via the LC via the CPU. This avoids the need to go to the RP for responses and plays in with the theme of distributed interface management.
2. ICMP (Layer 4) packets are received in the LC and they are sent via lookup through the IFBI into the CPU queues on the Sprayer. These packets are then sent to the CPU (via the Squid) and processed. The response is then sent through the Sprayer egress queues in order to be forwarded through the fabric. This is in case another application also needs the information (replicated through the fabric). Once through the fabric the packet is destined to the proper egress LC and through the proper sponge and control queue.
3. Routing flows are looked up in the IFIB and then sent to the output shaping queues (8000 queues) one of which is reserved for control packets. This is a non shaped queue and is simply serviced every time it is full.   high priority. The packet is then sent through the fabric on high priority queues into a set of CPU queues on the Sponge (similar to the Squid queues on the Sprayer), and then processes by the proper process, flow manager or the actual process. A response is sent back through the egress line card sponge and then out the Line card. The egress LC sponge has a special queue set aside to handle control packets. The queues in the Sponge are split into high priority, control and low priority packets, per egress port basis.
4. The PSE has a set of policers that are configured for rate limiting Layer 4, Layer 2 and routing packets. These are pre−set and change to be user configurable at a later date.

One of the most common problem with LPTS is packets that are dropped, when you attempt to ping the router. The LPTS policers are usually rate limiting these packets. This is the case in order to confirm:

```
RP/0/RP0/CPU0:ss01−crs−1_P1#ping 192.168.3.14 size 8000 count 100
Type escape sequence to abort.
Sending 100, 8000−byte ICMP Echos to 192.168.3.14, timeout is 2 seconds:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!!!!!!!!!!!!!.!!!!!!!!!!!
!!!!!!!!!!!!!!!!.!!!!!!!!!!!!!
Success rate is 97 percent (97/100), round−trip min/avg/max = 1/2/5 ms
RP/0/RP0/CPU0:ss01−crs−1_P1#show lpts pifib hardware entry statistics location 0/5/CPU0 | 

* − Vital; L4 − Layer4 Protocol; Intf − Interface;
DestAddr − Destination Fabric Address;
na − Not Applicable or Not Available

Local, Remote Address.Port          L4    Intf         DestAddr     Pkts/Drops
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− −−−−− −−−−−−−−−−−− −−−−−−−−−−−− −−− any
224.0.0.5 any                       any   PO0/5/1/0    0x3e         4/0
224.0.0.5 any                       any   PO0/5/1/1    0x3e         4/0
<further output elided>
```

IPsec

IP packets are inherently insecure. IPsec is a method used to protect the IP packets. CRS−1 IPsec is implemented in software forwarding path, therefore the IPsec session is terminated on the RP/DRP. A total number of 500 IPsec sessions per CRS−1 are supported. The number depends on CPU speed and allocated resources. There is no software limitation to this, only locally−sourced and locally−terminated traffic on RP are eligible for IPsec handling. Either IPsec transport mode or tunnel mode can be used for the type of traffic, though the former is preferred due to less overhead in IPsec processing.

R3.3.0 supports the encryption of BGP and OSPFv3 over IPsec.

Refer to Cisco IOS XR System Security Configuration Guide for more information on how to implement IPsec.

Note: IPsec requires crypto pie, for example, hfr−k9sec−p.pie−3.3.1.

Out of Band

Console and AUX Access

The CRS−1 RP/SCs have both a console and AUX port available for out of band management purposes, as well as a management Ethernet port for out−of−band via IP.

The console and AUX port of each RP/SCGE, two per chassis, can be connected to a console server. This means the single chassis system requires four console ports, and the multi−chassis systems require 12 ports plus two further ports for the Supervisor Engines on the Catalyst 6504−E.

The AUX port connection is important since it provides access to the IOS−XR kernel and can allow system recovery when this is not possible via the console port. Access via the AUX port is only available to users locally defined on the system, and only when the user has root−system or cisco−support level access. In addition the user must have a **secret** password defined.

Virtual Terminal Access

Telnet & Secure shell (SSH) can be used in order to reach the CRS−1 via the vty ports. By default both are disabled, and the user needs to explicitly enable them.

Note: IPsec requires crypto pie, for example, hfr−k9sec−p.pie−3.3.1.

First generate RSA and DSA keys as shown in this example in order to enable SSH:

```
RP/0/RP1/CPU0:Crs−1#crypto key zeroize dsa
% Found no keys in configuration.
RP/0/RP1/CPU0:Crs−1#crypto key zeroize rsa
% Found no keys in configuration.

RP/0/RP1/CPU0:Crs−1#crypto key generate rsa general−keys
The name for the keys will be: the_default
  Choose the size of the key modulus in the range of 360 to 2048 for your General Purpose
  Choosing a key modulus greater than 512 may take a few minutes.

How many bits in the modulus [1024]:
Generating RSA keys ...
Done w/ crypto generate keypair
[OK]

RP/0/RP1/CPU0:Crs−1#crypto key generate dsa
The name for the keys will be: the_default
  Choose the size of your DSA key modulus. Modulus size can be 512, 768, or 1024 bits. Cho
How many bits in the modulus [1024]:
Generating DSA keys ...
Done w/ crypto generate keypair
[OK]
```

```
!--- VTY access via SSH & telnet can be configured as shown here.

vty-pool default 0 4
ssh server
!
line default
 secret cisco
 users group root-system
 users group cisco-support
 exec-timeout 30 0
transport input telnet ssh
!
!
telnet ipv4 server
```

Related Information

- **Routers Support**
- **Technical Support & Documentation − Cisco Systems**

Updated: Nov 01, 2006 Document ID: 71770