ION Deployment Guide

Version 1.0

24 October 2011

Jane Marquart, NASA

Greg Menke, Columbus

Larry Shackelford, Microtel LLC

Scott Burleigh, Jet Propulsion Laboratory, California Institute of Technology

Contents

Acknowledgment

Some of the technology described in this Deployment Guide was developed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Overview

The effort required to deploy the Interplanetary Overlay Network (ION) software in an operational setting may vary widely depending on the scope of the deployment and the degree to which the required ION functionality coincides with the capability provided by default in the software as distributed. This effort will be expended in two general phases: initial ***infusion*** and ongoing ***operation***.

Infusion

Even in the best case, some minimal degree of configuration will be required. Many elements of ION behavior are managed at run time by decisions recorded in ION's protocol state databases, as populated by a variety of administration utility programs. Others are managed at compile time by means of compiler command-line switches selected when the software is built. These options are described in the ***Configuration*** section below.

In some cases, mission-specific behavior that goes beyond the options built into ION must be enabled during ION deployment. The intent of the ION design is to minimize – to eliminate, if possible – any need to modify ION source code in order to enable mission-specific behavior. Two general strategies are adopted for this purpose.

First, ION includes a number of conditionally defined functions that can be cleanly replaced with mission-specific alternative source code simply by setting a compiler command-line switch at build time. Setting such a switch causes the mission-specific source code, written in C, to be simply included within the standard ION source code at the time of compilation.

Second, more generally it is always possible to add new application executables, new startup/shutdown/monitor/control utilities or scripts, and even entirely new route computation systems, BP convergence-layer adapters, and/or LTP link service adapters without ever altering the distributed ION source code. Guidelines for making these kinds of modifications are described in the ***Adaptation*** section below.

Finally, in rare cases it may be necessary to operate ION in an operating-system environment to which it has not yet been ported. Guidance for porting ION to new platforms will be provided in a future edition of this Deployment Guide.

Operation

On an ongoing basis, an ION deployment may require reconfiguration from time to time and/or may require troubleshooting to resolve performance or stability problems. Some suggestions for reconfiguration and troubleshooting procedures are offered in the ***Operation*** section below.

Configuration

Building the "ici" package

Declaring values for the following variables, by setting parameters that are provided to the C compiler (for example, `-DFSWSOURCE` or `-DSM_SEMBASEKEY=0xff13`), will alter the functionality of ION as noted below.

PRIVATE_SYMTAB

This option causes ION to be built for VxWorks 5.4 or RTEMS with reliance on a small private local symbol table that is accessed by means of a function named `sm_FindFunction`. Both the table and the function definition are, by default, provided by the `symtab.c` source file, which is automatically included within the `platform_sm.c` source when this option is set. The table provides the address of the top-level function to be executed when a task for the indicated symbol (name) is to be spawned, together with the priority at which that task is to execute and the amount of stack space to be allocated to that task.

PRIVATE_SYMTAB is defined by default for RTEMS but not for VxWorks 5.4.

Absent this option, ION on VxWorks 5.4 must successfully execute the VxWorks `symFindByName` function in order to spawn a new task. For this purpose the entire VxWorks symbol table for the compiled image must be included in the image, and task priority and stack space allocation must be explicitly specified when tasks are spawned.

FSWLOGGER

This option causes the standard ION logging function, which simply writes all ION status messages to a file named `ion.log` in the current working directory, to be replaced (by `#include`) with code in the source file `fswlogger.c`. A file of this name must be in the inclusion path for the compiler, as defined by `-Ixxxx` compiler option parameters.

FSWCLOCK

This option causes the invocation of the standard `time` function within `getUTCTime` (in `ion.c`) to be replaced (by `#include`) with code in the source file `fswutc.c`, which might for example invoke a mission-specific function to read a value from the spacecraft clock. A file of this name must be in the inclusion path for the compiler.

FSWWDNAME

This option causes the invocation of the standard `getcwd` function within `cfdpInit` (in `libcfdpP.c`) to be replaced (by `#include`) with code in the source file `wdname.c`, which must in some way cause the mission-specific value of current working directory name to be copied into `cfdpdbBuf.workingDirectoryName`. A file of this name must be in the inclusion path for the compiler.

FSWSYMTAB

If the PRIVATE_SYMTAB option is also set, then the FSWSYMTAB option causes the code in source file `mysymtab.c` to be included in `platform_sm.c` in place of the default symbol table access implementation in `symtab.c`. A file named `mysymtab.c` must be in the inclusion path for the compiler.

FSWSOURCE

This option simply causes FSWLOGGER, FSWCLOCK, FSWWDNAME, and FSWSYMTAB all to be set.

GDSLOGGER

This option causes the standard ION logging function, which simply writes all ION status messages to a file named `ion.log` in the current working directory, to be replaced (by `#include`) with code in the source file `gdslogger.c`. A file of this name must be in the inclusion path for the compiler, as defined by `-Ixxxx` compiler option parameters.

GDSSOURCE

This option simply causes GDSLOGGER to be set.

ION_OPS_ALLOC=*xx*

This option specifies the percentage of the total non-volatile storage space allocated to ION that is reserved for protocol operational state information, i.e., is not available for the storage of bundles or LTP segments. The default value is 20.

ION_SDR_MARGIN=*xx*

This option specifies the percentage of the total non-volatile storage space allocated to ION that is reserved simply as margin, for contingency use. The default value is 20.

The sum of ION_OPS_ALLOC and ION_SDR_MARGIN defines the amount of non-volatile storage space that is sequestered at the time ION operations are initiated: for purposes of congestion forecasting and prevention of resource oversubscription, this sum is subtracted from the total size of the SDR "heap" to determine the maximum volume of space available for bundles and LTP segments. Data reception and origination activities fail whenever they would cause the total amount of data store space occupied by bundles and segments to exceed this limit.

USING_SDR_POINTERS

This is an optimization option for the SDR non-volatile data management system: when set, it enables the value of any variable in the SDR data store to be accessed directly by means of a pointer into the dynamic memory that is used as the data store storage medium, rather than by reading the variable into a location in local stack memory. Note that this option must **not** be enabled if the data store is

configured for file storage only, i.e., if the SDR_IN_DRAM flag was set to zero at the time the data store was created by calling `sdr_load_profile`. See the ionconfig(5) man page in Appendix A for more information.

NO_SDR_TRACE

This option causes non-volatile storage utilization tracing functions to be omitted from ION when the SDR system is built. It disables a useful debugging option but reduces the size of the executable software.

NO_PSM_TRACE

This option causes memory utilization tracing functions to be omitted from ION when the PSM system is built. It disables a useful debugging option but reduces the size of the executable software.

IN_FLIGHT

This option controls the behavior of ION when an unrecoverable error is encountered.

If it is set, then the status message "Unrecoverable SDR error" is logged and the SDR non-volatile storage management system is globally disabled: the current database access transaction is ended and (provided transaction reversibility is enabled) rolled back, and all ION tasks terminate.

Otherwise, the ION task that encountered the error is simply aborted, causing a core dump to be produced to support debugging.

SM_SEMKEY=0x*XXXX*

This option overrides the default value (0xee01) of the identifying "key" used in creating and locating the global ION shared-memory system mutex.

SVR4_SHM

This option causes ION to be built using svr4 shared memory as the pervasive shared-memory management mechanism. svr4 shared memory is selected by default when ION is built for any platform other than MinGW (for which File Mapping objects are used), VxWorks 5.4, or RTEMS. (For the latter two operating systems all memory is shared anyway, due to the absence of a protected-memory mode.)

POSIX1B_SEMAPHORES

This option causes ION to be built using POSIX semaphores as the pervasive semaphore mechanism. POSIX semaphores are selected by default when ION is built for RTEMS but are otherwise not used or supported; this option enables the default to be overridden.

SVR4_SEMAPHORES

This option causes ION to be built using svr4 semaphores as the pervasive semaphore mechanism. svr4 semaphores are selected by default when ION is built for any platform other than MinGW (for which Windows event objects are used), VxWorks 5.4 (for which VxWorks native semaphores are the default choice), or RTEMS (for which POSIX semaphores are the default choice).

SM_SEMBASEKEY=0x*XXXX*

This option overrides the default value (0xee02) of the identifying "key" used in creating and locating the global ION shared-memory semaphore database, in the event that svr4 semaphores are used.

SEMMNI=*xxx*

This option declares to ION the total number of svr4 semaphore sets provided by the operating system, in the event that svr4 semaphores are used. It overrides the default value, which is 128. (Changing this value typically entails rebuilding the O/S kernel.)

SEMMSL=*xxx*

This option declares to ION the maximum number of semaphores in each svr4 semaphore set, in the event that svr4 semaphores are used. It overrides the default value, which is 250. (Changing this value typically entails rebuilding the O/S kernel.)

SEMMNS=*xxx*

This option declares to ION the total number of svr4 semaphores that the operating system can support; the maximum possible value is SEMMNI x SEMMSL. It overrides the default value, which is 32000. (Changing this value typically entails rebuilding the O/S kernel.)

ION_NO_DNS

This option causes the implementation of a number of Internet socket I/O operations to be omitted for ION. This prevents ION software from being able to operate over Internet connections, but it prevents link errors when ION is loaded on a spacecraft where the operating system does not include support for these functions.

ERRMSGS_BUFSIZE=*xxxx*

This option set the size of the buffer in which ION status messages are constructed prior to logging. The default value is 4 KB.

SPACE_ORDER=*x*

This option declares the word size of the computer on which the compiled ION software will be running: it is the base-2 log of the number of bytes in an address. The default value is 2, i.e., the size of an address is 2^2 = 4 bytes. For a 64-bit machine, SPACE_ORDER must be declared to be 3, i.e., the size of an address is 2^3 = 8 bytes.

NO_SDRMGT

This option enables the SDR system to be used as a data access transaction system only, without doing any dynamic management of non-volatile data. With the NO_SDRMGT option set, the SDR system library can (and in fact must) be built from the `sdrxn.c` source file alone.

DOS_PATH_DELIMITER

This option causes ION_PATH_DELIMITER to be set to '\' (backslash), for use in construction path names. The default value of ION_PATH_DELIMITER is '/' (forward slash, as is used in Unix-like operating systems).

Building the "bp" package

Declaring values for the following variables, by setting parameters that are provided to the C compiler (for example, `-DION_NOSTATS` or `-DBRSTERM=60`), will alter the functionality of BP as noted below.

TargetFFS

Setting this option adapts BP for use with the TargetFFS flash file system on the VxWorks operating system. TargetFFS apparently locks one or more system semaphores so long as a file is kept open. When a BP task keeps a file open for a sustained interval, subsequent file system access may cause a high-priority non-BP task to attempt to lock the affected semaphore and therefore block; in this event, the priority of the BP task may automatically be elevated by the inversion safety mechanisms of VxWorks. This "priority inheritance" can result in preferential scheduling for the BP task – which does not need it – at the expense of normally higher-priority tasks, and can thereby introduce runtime anomalies. BP tasks should therefore close files immediately after each access when running on a VxWorks platform that uses the TargetFFS flash file system. The TargetFFS compile-time option ensures that they do so.

BRSTERM=*xx*

This option sets the maximum number of seconds by which the current time at the BRS server may exceed the time tag in a BRS authentication message from a client; if this interval is exceeded, the authentication message is presumed to be a replay attack and is rejected. Small values of BRSTERM are safer than large ones, but they require that clocks be more closely synchronized. The default value is 5.

ION_NOSTATS

Setting this option prevents the logging of bundle processing statistics in status messages.

KEEPALIVE_PERIOD=*xx*

This option sets the number of seconds between transmission of keep-alive messages over any TCP or BRS convergence-layer protocol connection. The default value is 15.

ION_BANDWIDTH_RESERVED

Setting this option overrides strict priority order in bundle transmission, which is the default. Instead, bandwidth is shared between the priority-1 and priority-0 queues on a 2:1 ratio whenever there is no priority-2 traffic.

Building the "ams" package

Defining the following macros, by setting parameters that are provided to the C compiler (for example, -DNOEXPAT or –DAMS_INDUSTRIAL), will alter the functionality of AMS as noted below.

NOEXPAT

Setting this option adapts AMS to expect MIB information to be presented to it in "amsrc" syntax (see the amsrc(5) man page in Appendix A) rather than in XML syntax, normally because the expat XML interpretation system is not installed. The default syntax for AMS MIB information is XML, as described in the amsxml(5) man page in Appendix A.

AMS_INDUSTRIAL

Setting this option adapts AMS to an "industrial" rather than safety-critical model for memory management. By default, the memory acquired for message transmission and reception buffers in AMS is allocated from limited ION working memory, which is fixed at ION start-up time; this limits the rate at which AMS messages may be originated and acquired. When –DAMS_INDUSTRIAL is set at compile time, the memory acquired for message transmission and reception buffers in AMS is allocated from system memory, using the familiar malloc() and free() functions; this enables much higher message traffic rates on machines with abundant system memory.

Building the "cfdp" package

Defining the following macro, by setting a parameter that is provided to the C compiler (i.e., –DTargetFFS), will alter the functionality of CFDP as noted below.

TargetFFS

Setting this option adapts CFDP for use with the TargetFFS flash file system on the VxWorks operating system. TargetFFS apparently locks one or more system semaphores so long as a file is kept open. When a CFDP task keeps a file open for a sustained interval, subsequent file system access may cause a high-priority non-CFDP task to attempt to lock the affected semaphore and therefore block; in this event, the priority of the CFDP task may automatically be elevated by the inversion safety mechanisms of VxWorks. This "priority inheritance" can result in preferential scheduling for the CFDP task – which does not need it – at the expense of normally higher-priority tasks, and can thereby introduce runtime anomalies. CFDP tasks should therefore close files immediately after each access when running on a VxWorks platform that uses the TargetFFS flash file system. The TargetFFS compile-time option assures that they do so.

Startup

ION requires several configuration settings to be defined at startup. Most notable are the settings for the Admin sections of ION. ION provides six admin utilities, namely bpadmin, dtn2admin, ipnadmin, ionadmin, ionsecadmin, and ltpadmin. If any of these are to be used at runtime, they will need to be configured. An admin configuration file is where the configuration commands are stored.

In the Linux environment, two different styles of configuration files are possible. The first style dictates that all configuration commands for all in-use admins will be stored in one file. This one file is sectioned off internally to separate the commands of each admin. The program named "ionstart" that installs as part of the official release is an AWK program that accepts this one configuration file's name as a parameter. It then parses through this file looking for sectioned-off areas for each possible admin and then uses the commands within these sections to configure each of the requested admins.

The other style dictates that each admin will have its own unique configuration file. The ionstart program supports command line switches and parameters indentifying each configuration file. The ION tutorial describes each in detail.

Runtime Parameters

Another way in which ION can be tweaked dynamically is through its runtime parameters. There are several values within ION that configured be set as ION starts. See the figure below for the identifier names and types of values accepted.

```
wmKey (integer)
wmSize (integer)
wmAddress (integer)
sdrName (string)
configFlags (bit pattern in integer form)
heapWords (integer)
heapKey (integer)
pathName (string)
```

The manner in which these values get read into ION is also file dependent. One of the more prominent admin commands is the ionadmin's "1" (the numeral one) command. The first parameter to this command is required and is a numeric value that represents the node number of the DTN node being configured. The second parameter to this command is an optional one and, if present, holds the full pathname of an onboard file of runtime parameter values. On both Linux and VxWorks systems, this filename should NOT be enclosed in any type of quotation marks.

If specified and located, this file is a sequentially processed text file with 2 fields per line. The first field on each line holds one of the parameter identifier text strings as shown above. The second field holds the value that will be placed into the identified parameter. Make sure that the data type specified in the second field matches the type expected.

For documentation on these parameters, see the ionconfig(5) man page.

The "configFlags" entry controls several features of the SDR in memory. There are several flags of interest:

```
#define SDR_IN_DRAM     1       /*      Write to & read from memory.    */
#define SDR_IN_FILE     2       /*      Write file; read file if nec.   */
#define SDR_REVERSIBLE  4       /*      Transactions may be reversed.   */
```

SDR_IN_DRAM is required for normal ION operation and should virtually always be specified.

When SDR_REVERSIBLE is specified, SDR transactions that fail (e.g., due to memory allocation failure) are rolled back, allowing transactions to fail gracefully without corrupting the ION data bases. If the flag is not supplied, failed transactions will cause an immediate exit of the task receiving the failure. This feature is intended as an aid to debugging, so in operations ION should be normally operated with reversible transactions. When transaction reversibility is enabled, ION creates & manages a log file in the directory named by "pathName" which must be writable by ION and which tracks the SDR changes and supports rollback to the last consistent state. The filesystem for this directory should be high-performance but needn't be nonvolatile; a ramdisk is ideal. The maximum size of the logfile is dependent upon the largest transaction in the SDR, and is therefore of a size in the same order of magnitude as the largest bundle. NOTE that if the directory named by “pathname” does not exist then transaction reversibility will be disabled automatically; a message to this effect will be written to the ION log file.

When SDR_IN_FILE is specified, ION creates a file in the "pathName" directory, which is maintained as a copy of the SDR heap in DRAM; whenever the SDR heap in memory is modified, the changes are also written to the sdr heap file. Thus the heap file is always the same size as the in-memory heap. Again, if the directory named by “pathname” does not exist then retention of the ION SDR heap in a file will be disabled automatically; a message to this effect will be written to the ION log file.

For documentation on the admin commands, see the man pages. The man page names are in the form of xxxrc, where xxx gets replaced by the specific admin (bp, dtn2, ion, ionsec, ipn, ltp). The directories in which to find these files are: ./ici/doc/pod5, ./ltp/doc/pod5 and ./bp/doc/pod5.

Multi-node Operation

Normally the instantiation of ION on a given computer establishes a single ION node on that computer, for which hard-coded values of wmKey and sdrName (see ionconfig(5)) are used in common by all executables to assure that all elements of the system operate within the same state space. For some purposes, however, it may be desirable to establish multiple ION nodes on a single workstation. (For example, constructing an entire self-contained DTN network on a single machine may simplify some kinds of regression testing.) ION supports this configuration option as follows:

- Multi-node operation on a given computer is enabled if and only if the environment variable ION_NODE_LIST_DIR is defined in the environment of every participating ION process. Moreover, the value assigned to this variable must be the same text string in the environments of all participating ION processes. That value must be the name (preferably, fully qualified) of the directory in which the ION multi-node database file “ion_nodes” will reside.

- The definition of ION_NODE_LIST_DIR makes it possible to establish up to one ION node per directory rather than just one ION node on the computer. When **ionadmin** is used to establish a node, the `ionInitialize()` function will get that node's wmKey and sdrName from the .ionconfig file, use them to allocate working memory and create the SDR database, and then write a line to the ion_nodes file noting the nodeNbr, wmKey, sdrName, and wdName for the node it just initialized. wdName is the current working directory in which **ionadmin** was running at the time it called `ionInitialize()`; it is the directory within which the node resides.
- This makes it easy to connect all the node's daemon processes – running within the same current working directory – to the correct working memory partition and SDR database: the `ionAttach()` function simply searches the ion_nodes file for a line whose wdName matches the current working directory of the process that is trying to attach, then uses that line's wmKey and sdrName to link up.
- It is also possible to initiate a process from within a directory other than the one in which the node resides. To do so, define the additional environment variable ION_NODE_WDNAME in the shell from which the new process is to be initiated. When `ionAttach()` is called it will first try to get "current working directory" (for ION attachment purposes **only**) from that environment variable; only if ION_NODE_WDNAME is undefined will it use the actual cwd that it gets from calling igetcwd().

Shutdown

As ION was initially designed, the Linux command "ionstop" called up a bash script. This script in turn called each of the "admins" and instructed each subsystemto exit by supplying the dummy command file name ".". Once all of the "admins" had exited, another shell script was executed. This "killm" script attempted to incrementally kill each of the Linux processes by name.

Example Configuration Files

ION Node Number Cross-reference

When you define a DTN node, you do so using ionadmin and its Initialize command (using the token '1'). This node is then referenced by its node number throughout the rest of the configuration file. Instances where this node number is defined (on line number 2) and referenced are highlighted below in _magenta_. The node numbers of other nodes referenced below (but defined elsewhere) are highlighted in _green_ and _cyan_.

```
 1  ## begin ionadmin
 2  1   1  /home/spwdev/cstl/ion-configs/23/badajoz/3node-udp-ltp/badajoz.ionconfig
 3
 4  s
 5
 6  a  contact  +1  +86400   25   25  50000000
 7  a  contact  +1  +84600   25  101  50000000
 8  a  contact  +1  +84600   25    1  50000000
 9
10  a  contact  +1  +86400  101   25  50000000
```

```
11 a contact +1 +86400  101  101 50000000
12 a contact +1 +86400  101    1 50000000
13
14 a contact +1 +86400    1   25 50000000
15 a contact +1 +86400    1  101 50000000
16 a contact +1 +86400    1    1 50000000
17
18
19 a range +1 +86400   25   25 10
20 a range +1 +86400   25  101 10
21 a range +1 +86400   25    1 10
22
23 a range +1 +86400  101   25 10
24 a range +1 +86400  101  101 10
25 a range +1 +86400  101    1 10
26
27 a range +1 +86400    1   25 10
28 a range +1 +86400    1  101 10
29 a range +1 +86400    1    1 10
30
31
32 m production  50000000
33 m consumption 50000000
34
35 ## end ionadmin
36 ################################################################################
37 ## begin ltpadmin
38 1 32 300000
39
40 a span  25 1 1 1400 1400 1 'udplso 192.168.1.25:1113'
41 a span 101 1 1 1400 1400 1 'udplso 192.168.1.101:1113'
42 a span   1 1 1 1400 1400 1 'udplso 192.168.1.1:1113'
43
44 s 'udplsi 192.168.1.1:1113'
45
46 ## end ltpadmin
47 ################################################################################
48 ## begin bpadmin
49 1
50
51 a scheme ipn 'ipnfw'   'ipnadminep'
52
```

```
53  a  endpoint  ipn:1.0  q
54  a  endpoint  ipn:1.1  q
55  a  endpoint  ipn:1.2  q
56
57  a  protocol  ltp  1400  100
58  a  protocol  tcp  1400  100
59
60  a  outduct  ltp   25          ltpclo
61  a  outduct  ltp  101          ltpclo
62  a  outduct  ltp   1          ltpclo
63
64  a  induct   ltp   1          ltpcli
65
66  s
67
68  ## end bpadmin
69  ############################################################################
70  ##  begin  ipnadmin
71  a  plan   25  ltp/25
72  a  plan  101  ltp/101
73  a  plan   1  ltp/1
74
75  ##  end  ipnadmin
76
```

IPN Parameters Cross-reference

The IPN scheme establishes static default routing rules for forwarding bundles to specified destination nodes. Any transmission using the IPN rules will have endpoints IDs of this form:

ipn:*nodenumber*. *servicenumber*.

The Add Scheme command on line 51 below specifies that the IPN routing rules will be available, and that the naming scheme for endpoint IDs will be shown in lines 53 thru 55.

The two remaining parameters on this command are used to define the software functions that will act as data forwarder and endpoint data receiver.

The bpadmin Add Scheme command

a scheme *scheme_name forwarder_command admin_app_command*

The add scheme command. This command declares an endpoint naming "scheme" for use in endpoint IDs, which are structured as URIs: *scheme_name*:*scheme-specific_part*. *forwarder_command* will be executed when the scheme is started on this node, to initiate operation of a forwarding daemon for this scheme. *admin_app_command* will also be executed when the scheme is started on this node, to

initiate operation of a daemon that opens an administrative endpoint identified within this scheme so that it can receive and process custody signals and bundle status reports.

Starting at line 71, the egress plans are defined. These determine the routes that data must follow as they leave the current node.

The ipndmin Add Plan command

a plan *node_nbr default_duct_expression*

The add plan command. This command establishes an egress plan for the bundles that must be transmitted to the neighboring node identified by node_nbr. A general plan must be in place for a node before any more specific rules are declared.

Each duct expression is a string of the form

protocol_name/*outduct_name*[,*destination_induct_name*]

signifying that the bundle is to be queued for transmission via the indicated convergence layer protocol outduct. *destination_induct_name* must be provided when the indicated outduct is "promiscuous", i.e., not configured for transmission only to a single neighboring node; this is protocol-specific.

The configuration below defines three DTN nodes with the following connections.

```
 1  ## begin ionadmin
 2  1  1  /home/spwdev/cstl/ion-configs/23/badajoz/3node-udp-ltp/badajoz.ionconfig
 3
 4  s
 5
 6  a  contact  +1  +86400    25    25  50000000
 7  a  contact  +1  +84600    25   101  50000000
 8  #a  contact  +1  +84600    25     1  50000000
 9
10  a  contact  +1  +86400   101    25  50000000
11  a  contact  +1  +86400   101   101  50000000
12  a  contact  +1  +86400   101     1  50000000
13
14  #a  contact  +1  +86400     1    25  50000000
15  a  contact  +1  +86400     1   101  50000000
16  a  contact  +1  +86400     1     1  50000000
17
18
19  a  range  +1  +86400    25    25  10
20  a  range  +1  +86400    25   101  10
```

```
#a range +1 +86400   25    1  10

a range +1 +86400  101   25  10
a range +1 +86400  101  101  10
a range +1 +86400  101    1  10

#a range +1 +86400    1   25  10
a range +1 +86400    1  101  10
a range +1 +86400    1    1  10


m production  50000000
m consumption 50000000

## end ionadmin
################################################################################
## begin ltpadmin
1 32 300000

a span  25 1 1 1400 1400 1 'udplso 192.168.1.25:1113'
a span 101 1 1 1400 1400 1 'udplso 192.168.1.101:1113'
a span   1 1 1 1400 1400 1 'udplso 192.168.1.1:1113'

s 'udplsi 192.168.1.1:1113'

## end ltpadmin
################################################################################
## begin bpadmin
1

a scheme ipn 'ipnfw'   'ipnadminep'

a endpoint ipn:1.0 q
a endpoint ipn:1.1 q
a endpoint ipn:1.2 q

a protocol ltp 1400 100
a protocol tcp 1400 100

a outduct ltp  25   ltpclo
a outduct ltp 101   ltpclo
a outduct ltp   1   ltpclo
```

```
63
64 a  induct   ltp   1     ltpcli
65
66 s
67
68 ## end bpadmin
69 ############################################################################
70 ##  begin  ipnadmin
71 a  plan   25  ltp/25
72 a  plan  101  ltp/101
73 a  plan    1  ltp /1
74
75 ##  end  ipnadmin
76
```

LTP Parameters Cross-reference

When configured and used properly, the ltpadmin utility allows the features of the LTP protocol to become available. For details of the LTP protocol, see RFC 5325.

The first command that must be issued to ltpadmin is the Initialize command (see line number 38 below, the command token is the '1' (one)). The two numeric parameters to this command are *est_max_export_sessions* and *database_bytes_needed*.

The ltpadmin Initialize command

This command reserves database_bytes_needed bytes of storage in ION's SDR database for management of LTP's queuing and retransmission buffers. It uses *est_max_export_sessions* to configure the hash table it will use to manage access to export transmission sessions that are currently in progress. (For optimum performance, *est_max_export_sessions* should normally equal or exceed the summation of *max_export_sessions* over all spans as discussed below.)

Appropriate values for these parameters and for the parameters configuring each "span" of potential LTP data exchange between the local LTP and neighboring engines are non-trivial to determine. See the ION LTP configuration spreadsheet and accompanying documentation for details.

- Essentially, the "max export sessions" must be >= the total number of export sessions on all the spans. If it is expected that new spans will be added during an ION session, then max export sessions figure should be large enough to cover the maximum # of sessions possible.

- The "database bytes needed" field must specify a value sufficiently large to support the LTP management information for all traffic on all the spans, plus the traffic itself when transmitted and received blocks are so small that they are recorded directly within SDR heap space. As the # of spans, export sessions and aggregation parameters increase, this value increases commensurately. If this value is too small ION will record a warning at LTP startup – the warning should be considered a fatal error and the configuration adjusted.

Next to be defined are the Spans. They define the interconnection between two LTP engines. There are many parameters associated with the Spans.

The ltpadmin Add Span command

a span *peer_engine_nbr max_export_sessions max_import_sessions max_segment_size aggregation_size_limit aggregation_time_limit 'LSO_command'* [*queuing_latency*]

The "add span" command. This command declares that a span of potential LTP data interchange exists between the local LTP engine and the indicated (neighboring) LTP engine.

The *max_segment_size* and the *aggregation_size_limit* are expressed as numbers of bytes of data. *max_segment_size* limits the size of each of the segments into which each outbound data block will be divided; typically this limit will be the maximum number of bytes that can be encapsulated within a single transmission frame of the underlying link service. *max_segment_size* specifies the largest LTP segment that this span will produce.

aggregation_size_limit limits the number of LTP service data units (e.g., bundles) that can be aggregated into a single block: when the sum of the sizes of all service data units aggregated into a block exceeds this limit, aggregation into this block must cease and the block must be segmented and transmitted. When numerous small bundles are outbound, they are aggregated into a block of this size instead of being sent individually.

aggregation_time_limit alternatively limits the number of seconds that any single export session block for this span will await aggregation before it is segmented and transmitted, regardless of size. The aggregation time limit prevents undue delay before the transmission of data during periods of low activity. When few small bundles are outbound, they are collected until this time limit is met, whereupon the aggregated quantity is sent as a single, larger block.

max_export_sessions constitutes the size of the local LTP engine's retransmission "window" for this span. The retransmission windows of the spans impose flow control on LTP transmission, preventing the allocation of all available space in the ION node's data store to LTP transmission sessions.

The *max_import_sessions* parameter is simply the neighboring engine's own value for the corresponding export session parameter.

LSO_command is script text that will be executed when LTP is started on this node, to initiate operation of a link service output task for this span. Note that *peer_engine_nbr* will automatically be appended to *LSO_command* by ltpadmin before the command is executed, so only the link-service-specific portion of the command should be provided in the *LSO_command* string itself.

queuing_latency is the estimated number of seconds that we expect to lapse between reception of a segment at this node and transmission of an acknowledging segment, due to processing delay in the node. (See the 'm ownqtime' command below.) The default value is 1.

If *queuing_latency* is a negative number, the absolute value of this number is used as the actual queuing latency and **session purging** is enabled; otherwise session purging is disabled. If session purging is enabled for a span then at the end of any period of transmission over this span all of the span's export sessions that are currently in progress are automatically canceled. Notionally this forces re-forwarding of the DTN bundles in each session's block, to avoid having to wait for the restart of transmission on this span before those bundles can be successfully transmitted.

Additional notes:

- A "session block" is filled by outbound bundles until its aggregation size limit is reached, or its aggregation time is reached, whereupon it is output as a series of segments (of size bounded by *max_segment_size*). This series of segments is reliably transferred via a LTP protocol handshake with the remote node, one handshake per block (more if retries are needed to recover missing segments). By adjusting the size of the session block, the rate of arrival of response segments from the remote node can be controlled. Assuming a bundle rate sufficient to fill the session block, a large session block size means a lot of LTP segments per handshake (good for a high-rate return, low-rate forward link situation). A small session block size means the # of segments per handshake is smaller and the LTP protocol will complete the block transfer more quickly because the # of segment retries is generally smaller.

- A good starting point for a configuration is to set aggregation size limit to the number of bytes that will typically be transmitted in one second, so that blocks are typically clocked out about once per second. The maximum number of export sessions then should be at least the total number of seconds in the round-trip time for traffic on this LTP span, to prevent transmission from being blocked due to inability to start another session while waiting for the LTP acknowledgment that can end one of the current sessions.

- Multiple session blocks permit bundles to stream; while one session block is being transmitted, a second can be filled (and itself transmitted) before the first is completed. By increasing the number of blocks, high latency links can then be filled to capacity (provided there is adequate bandwidth available in the return direction for the LTP handshakes). It is desirable to reduce the max_export_sessions to a value where "most" of the sessions are employed because each session allocates an increment of buffer memory from the SDR whether it is used or not.

- When a session block is transmitted, it is emitted as a series of back-to-back LTP segments that are not metered in any way. The underlying link layer must be prepared to queue the entire set and forward them at the actual link rate. The udplso task has an optional rate-limit parameter which will meter the segments if packet loss is experienced as a consequence of the burst.

- Extreme asymmetries of LTP segments per handshake become sensitive to BER & packet loss issues which will tend to increase the number of handshakes per session block. In the CSTL lab we have had reasonable luck operating LTP at asymmetries of approx 1000:1, where the forward link consumption is constrained to approximately 1 kilobit/sec.

The ltpadmin Start command

s '*LSI command*'

This command starts link service output tasks for all LTP spans (to remote engines) from the local LTP engine, and it starts the link service input task for the local engine.

The sole command on line number 44 below starts two main operations within LTP. The first of these operations starts all of the link service output tasks, the ones defined for each LTP span (see the *LSO_command* parameter of the Add Span command). In this example, each task instantiates the same function (named 'udplso'). Each 'udplso' needs a destination for its transmissions and these are defined as hostname or IP Address (192.168.1.*) and port number (1113, the pre-defined default port number for all LTP traffic).

The second operation started by this command is to instantiate the link service input task. In this instance, the task is named "udplsi". It is through this task that all LTP input traffic will be received. Similar to the output tasks, the input task also needs definition of the interface on which LTP traffic will arrive, namely hostname or IP address (192.168.1.1) and port number (1113).

Once the LTP engine has been defined, initialized and started, we need a definition as to how data gets routed to the Convergence Layer Adaptors. Defining a protocol via bpadmin is the first step in that process.

The bpdmin Add Protocol command

a protocol *protocol_name payload_bytes_per_frame overhead_bytes_per_frame* [*nominal_data_rate*]

The "add protocol" command. This command establishes access to the named convergence layer protocol at the local node. The *payload_bytes_per_frame* and *overhead_bytes_per_frame* arguments are used in calculating the estimated transmission capacity consumption of each bundle, to aid in route computation and congestion forecasting.

The optional *nominal_data_rate* argument overrides the hard-coded default continuous data rate for the indicated protocol, for purposes of rate control. For all CL protocols other than LTP, the protocol's applicable nominal continuous data rate is the data rate that is always used for rate control over links served by that protocol; data rates are not extracted from contact graph information. This is because only the LTP induct and outduct throttles can be dynamically adjusted in response to changes in data rate between the local node and its neighbors, because (currently) there is no mechanism for mapping neighbor node number to the duct name for any other CL protocol. For LTP, duct name is simply LTP engine number which, by convention, is identical to node number. For all other CL protocols, the nominal data rate in each induct and outduct throttle is initially set to the protocol's configured nominal data rate and is never subsequently modified.

Once the protocol has been defined, it can now be used to define ducts; both inducts and outducts as seen in lines 76 thru 80 below. The Add "duct" commands associate a protocol (in this case, LTP) with individual node numbers (in this case, 25, 101 and 1) and a task designed to handle the appropriate

Convergence Layer output operations. A similar scenario applies for the induct where the LTP protocol and node number 13 get connected with “ltpcli” as the input Convergence Layer function.

The bpadmin Add Outduct and Add Induct commands

a outduct *protocol_name* duct_name ‘*CLO_command*’

The “add outduct” command. This command establishes a "duct" for transmission of bundles via the indicated CL protocol. The duct’s data transmission structure is serviced by the "outduct" task whose operation is initiated by *CLO_command* at the time the duct is started.

a induct *protocol_name duct_name* ‘*CLI_command*’

The “add induct” command. This command establishes a "duct" for reception of bundles via the indicated CL protocol. The duct’s data acquisition structure is used and populated by the "induct" task whose operation is initiated by *CLI_command* at the time the duct is started.

Once all of this has been defined, the last piece needed is the egress plan -- namely how do packets get routed to their final destination.

As you can see from line numbers 6 thru 29, the only network neighbor to node 1 is node 101. Node 25 has not been defined (because the commands in lines 8, 14, 21 and 27 have been commented). In line numbers 15 and 16, we see that the only destinations for data beginning at node 1 are nodes 101 and 1 (a loopback as such). Therefore, in order to get data from node 1 to node 25, our only choice is to send data to node 101. Out best hope of reaching node 25 is that the configurations for node 101 define a connection to node 25 (either a one-hop direct connection, or more multi-hop assumptions). This is where egress plans come into play.

On line numbers 87 thru 89, this configuration defines the only choices that can be made regarding destinations. For a destination of node 25, which is not a neighbor, all node 1 can do is pass the data to its only neighbor, namely node 101; the “group” command enables this operation. For destinations of nodes 101 and 1, the scenario is pretty simple.

The ipndmin Add Group command

a group *first_node_nbr last_node_nbr gateway_endpoint_ID*

The “add group” command. This command establishes a “group” for static routing. A group is a range of node numbers identifying a set of nodes for which defined default routing behavior is established. Whenever a bundle is to be forwarded to a node whose number is in the group’s node number range **and** it has not been possible to compute a dynamic route to that node from the contact schedules that have been provided to the local node **and** that node is not a neighbor to which the bundle can be directly transmitted, BP will forward the bundle to the gateway node associated with this group.

The ipndmin Add Plan command

a plan *node_nbr default_duct_expression*

The “add plan” command. This command establishes an egress plan for the bundles that must be transmitted to the neighboring node identified by *node_nbr*. A general plan must be in place for a node before any more specific rules are declared.

Each duct expression is a string of the form

"*protocol_name*/*outduct_name*[,*destination_induct_name*]"

signifying that the bundle is to be queued for transmission via the indicated convergence layer protocol outduct. *destination_induct_name* must be provided when the indicated outduct is "promiscuous", i.e., not configured for transmission only to a single neighboring node; this is protocol-specific.

The duct expression used in these examples has “ltp” being the protocol name and 101 and 1 being the outduct names.

The configuration below defines three DTN nodes with the following connections.

```
 1  ## begin ionadmin
 2  1  1  /home/spwdev/cstl/ion-configs/23/badajoz/3node-udp-ltp/badajoz.ionconfig
 3
 4  s
 5
 6  a  contact  +1  +86400    25    25  50000000
 7  a  contact  +1  +84600    25   101  50000000
 8  #a  contact  +1  +84600    25     1  50000000
 9
10  a  contact  +1  +86400   101    25  50000000
11  a  contact  +1  +86400   101   101  50000000
12  a  contact  +1  +86400   101     1  50000000
13
14  #a  contact  +1  +86400     1    25  50000000
15  a  contact  +1  +86400     1   101  50000000
16  a  contact  +1  +86400     1     1  50000000
17
18
19  a  range  +1  +86400    25    25  10
20  a  range  +1  +86400    25   101  10
21  #a  range  +1  +86400    25     1  10
22
23  a  range  +1  +86400   101    25  10
24  a  range  +1  +86400   101   101  10
25  a  range  +1  +86400   101     1  10
```

```
26
27 #a range +1 +86400    1   25  10
28 a range +1 +86400    1   101  10
29 a range +1 +86400    1    1  10
30
31
32 m production   50000000
33 m consumption  50000000
34
35 ## end ionadmin
36 ###############################################################################
37 ## begin ltpadmin
38 1 32 300000
39
40 a span   25  1  25000  1  25000  1400  1400  1  'udplso 192.168.1.25:1113'
41 a span  101  1  25000  1  25000  1400  1400  1  'udplso 192.168.1.101:1113'
42 a span    1  1  25000  1  25000  1400  1400  1  'udplso 192.168.1.1:1113'
43
44 s 'udplsi 192.168.1.1:1113'
45
46 ## end ltpadmin
47 ###############################################################################
48 ## begin bpadmin
49 1
50
51 a scheme ipn 'ipnfw'   'ipnadminep'
52
53 a endpoint ipn:1.0 q
54 a endpoint ipn:1.1 q
55 a endpoint ipn:1.2 q
56
57 a protocol ltp 1400 100
58 a protocol tcp 1400 100
59
60 #a outduct ltp   25   ltpclo
61 a outduct ltp  101   ltpclo
62 a outduct ltp    1   ltpclo
63
64 a induct   ltp   1    ltpcli
65
66 s
67
```

```
68  ## end bpadmin
69  ##############################################################################
70  ##  begin  ipnadmin
71  #a  plan   25  ltp/25
72  a  plan  101  ltp/101
73  a  plan    1  ltp /1
74  a group 25 25 101
75  ##  end  ipnadmin
76
```

Adaptation

Error Logging

ION contains a flexible system that allows its code to display errors in several different ways. At the core of this system is a typedef that defines a data type named "Logger" (with upper case "L") that is a function variable that accepts a character pointer (string) parameter and returns a value of type void.

```
typedef void                    (* Logger)(char *);
```

In ION, there is one variable defined to be of this type. Its identifier is "logger" (with lower case "L") and it is initialized to a value of "logToStdout". The function "logToStdout" is defined and its contents cause the string parameter to be printed to the stdout device. Therefore, any call to the function variable "logger" will have same effects as a call to the function "logToStdout".

However, remember that "logger" is a variable and is allowed to change its value to that of other functions that accept string parameters and return void. This is how ION allows for flexibility in logging errors.

At startup, ION makes a call to "ionRedirectMemos". This function makes a call to "setLogger" which eventually changes the value of the "logger" variable. The new value of the variable named "logger" is "writeMemoToIonLog". This function writes strings to a file named "ion.log".

It is through this mechanism that any calls to the functions "writeMemo", "writeMemoNote" or "writeErrMemo" eventually pass their parameters to the function "writeMemoToIonLog". This is how the Linux-based ION's operate.

Checkout the FSWLOGGER macro option as documented in section 2.1.1 of the Design Guide.

Memory Allocation

What types of memory does ION use and how is memory allocated/controlled?

For an introductory description of the memory resources used by ION, see Section 1.5 of the ION Design and Operation guide entitled “Resource Management in ION”.

Section 1.5 of the Design and Operation guide makes reference to parameters called “wmSize” and “heapWords”. Discussion on these and all of the parameters can be found in this document under the section entitled “Runtime Parameters”.

As currently configured, ION allocates its large blocks of memory via calls to malloc. The call tree to get to the spot of the actually call is as follows: (the indicated line numbers are approximate)

Call Tree for param “wmSize”	Call Tree for param “heapWords”
ionInitialize, ionadmin.c, line 161	ionInitialize, ionadmin.c, line 161
_ionwm, ion.c, line 667	sdr_load_profile – ion.c, line 573
memmgr_open, ion.c, line 154	sm_ShmAttach – sdrxn.c, line 1118
sm_ShmAttach – memmgr.c, line 304	acquireSystemMemory – platform_sm.c, line 91
acquireSystemMemory – platform_sm.c, line 91	memalign – platform.c, line 1294
memalign – platform.c, line 1294	malloc – platform.c, line 422
malloc – platform.c, line 422	

Should the need ever arise to place these large blocks of memory at known, fixed addresses, it would be possible to modify the function memalign, in the file platform.c , near line 422. A better approach would be to use ION’s sm_ShmAttach() function to create a shared-memory attachment to each pre-allocated memory block and pass the applicable shared-memory key values to ION at startup, in the “heapKey” and/or “wmKey” runtime parameters.

Any code that references the function “sm_ShmAttach” will be looking to acquire some block of memory. These would include the Space Management Trace features and standalone programs such as “file2sm”, “sm2file” and “smlistsh”.

Operation

ION is generally optimized for continuous operational use rather than research. In practice, this means that a lot more attention, both in the code and in the documentation, has been paid to the care and feeding of an existing ION-based network than to the problem of setting up a new network in the first place. (The unspoken expectation is that you're only going to do it once anyway.)

Unfortunately this can make ION somewhat painful for new users to work with. The notes in this section are aimed at reducing this pain, at least a little.

"Wrong profile for this SDR"

ION is based on shared access to a common database in memory (and/or in a file), and the objects in that database are intended to persist across multiple restarts of network activity in a continuously operational network. That's okay for Space Station operations, but it's not helpful while you're still struggling to get the network running in the first place. A key concept:

> Each time you run the standard **ionstart** script provided with ION, you are creating a new network from scratch. To minimize confusion, be sure to clear out the old database first.

In most cases the **killm** script should do this for you. Invoke it once on every machine you're using in your network. To verify that you're starting from a clean slate, run the **ipcs** command after **killm**: the list of Semaphore Arrays should be empty. If it's not, you've got one or more leftover processes from the previous network still running; use **ps ax** to find them and **kill -9** to get rid of them. The process names to look for are:

- Most names that start with "ion", "bp", "brs", "stcp", "dgr", "ltp", "cfdp", or "ams".
- udpcla, tcpcli, tcpclo, rfxclock.

If you don't wipe out the old system before trying to start the new one, then either you will pick up where you left off in testing the old system (and any endpoints, ducts, etc. you try to add will be rejected as duplicates) or – in the event that you have changed something fundamental in the configuration, or are using an entirely different configuration file – you'll see the "Wrong profile for this SDR" message and won't be able to continue at all.

An additional wrinkle: if you configure ION to manage your ION database in a file as well as (or instead of) managing it in shared memory, then in addition to calling killm to destroy the semaphores and the copy of the database that resides in shared memory, you also need to delete the database file; this destroys the copy of the database that resides in the file system. If the database isn't deleted, then when you restart ION using your standard configuration file the file-system copy of the database will automatically be reloaded into shared memory and all the config file commands that create new schemes, endpoints, etc. will fail, because they're still in the database that you were using before.

Another habit that can be helpful: whenever you restart ION from scratch, delete all the ion.log files in all of the directories in which you're configuring your ION nodes. This isn't mandatory – ION will happily append new log messages to existing log files, and the messages are time-tagged anyway, so it's always

possible to work out what happened when. But starting fresh with new log files removes a lot of clutter so that it's easy to see exactly what's happening in this particular iteration of your network research. ION will create new log files automatically if they don't exist; if there's something particularly interesting in the log from a prior system, copy that log file with a different name so you can come back to it if you need to.

"No such directory; disabling heap residence in file…"

This message just means that the directory whose name you've provided as the value of *pathName* in the ION configuration file does not exist, and therefore the ION operations that rely on being able to write files in that directory are disabled. It's strictly informative; nearly everything in ION will work just fine even if it's printed every time you run.

But if you do care about transaction reversibility, for example, or if you just want to get rid of the annoying message, simply create the directory that is named in *pathName* (it can be any path name you like) and make sure it's world-writable. The ionconfig(5) man page discusses this parameter and others that affect the fundamental character of the system you're configuring.

"Can't find ION security database"

These messages are just warnings, but they are annoying. We're still struggling to work out a way to support bundle security protocol as fully and readily as possible but still let people run ION without it, if they want, without too much hassle.

For now, the best answer might be to insert the following lines into each host.rc file immediately after the "##end ionadmin" line. They should create an empty ION security database on each host, which should shut down all those warnings:

```
## begin ionsecadmin
1
## end ionsecadmin
```

Clock sync

Several key elements of ION (notably LTP transmission and bundle expiration) rely on the clocks of all nodes in the network being synchronized to within a few seconds. NTP is a good way to accomplish this, if you've got access to an NTP server. If you can't get your clocks synchronized, stick to the TCP or UDP convergence-layer adapters, don't count on using contact graph routing, and use long lifetimes on all bundles to prevent premature bundle expiration.

Node numbers

In ION we always use the same numeric value for LTP engine number and BP node number – and for CFDP entity number as well. The idea is that a given ION node has a single identifying number, which by convention we use wherever a protocol endpoint identifier is needed for any local protocol agent. This is not a DTN or CCSDS requirement, but it doesn't violate any of the protocol specifications and it does marginally simplify both implementation and configuration.

Duct names

The bprc(5) man page explains the general format of the commands for adding convergence-layer inducts and outducts, but it doesn't provide the syntax for duct names, since duct name syntax is different for different CL protocols. Here's a summary of duct name syntax for the CL protocols supported as of ION 2.5.3:

- LTP: the duct name is simply the engine number, which (again) in ION is always the same as the BP node number. E.g., "9". The induct name is the number of the local node. One outduct is needed for each node to which bundles will be sent by LTP (including the local node, if you want to do BP loopback transmission), naming the LTP induct of the receiving node.
- DGR, TCP, and STCP ("simple TCP"): the duct name format is *hostname*[:*portnbr*], where *portnbr* defaults to "4546" if omitted. On most systems, an IP address can be provided instead of a DNS name as *hostname*. The induct name identifies the local node's machine. One outduct is needed for each node to which bundles will be sent, naming the induct of the receiving node.
- UDP: in ION, UDP is a "promiscuous" convergence-layer protocol. The name of a UDP induct has the same format as the names of TCP and STCP inducts, but the name of the sole UDP outduct is always simply "*". When ipn-scheme routing "plans" must direct bundles to nodes reachable by UDP, the duct expression format is udp/*,*hostname*[:*portnbr*] where *hostname*[:*portnbr*] identifies the UDP induct of the receiving node.
- BRSS (bundle relay service – server): BRSS is another promiscuous CL protocol. The induct and sole outduct have the same name, in the format *hostname*[:*portnbr*] identifying the local machine. ipn-scheme routing plans must provide the node numbers of the receiving nodes: brss/*hostname*[:*portnbr*],*nodenbr*.
- BRSC (bundle relay service – client): the induct and sole outduct have the same name, of the form *hostname*[:*portnbr*]_*nodenbr*, where *hostname*[:*portnbr*] is the BRS server's induct name and *nodenbr* is the local node's BP node number.

Config file pitfalls to watch for

Here are some other points to bear in mind as you debug your ION node configuration:

- The ranges between all pairs of neighboring nodes are required for the correct operation of contact graph routing. The canonical form of a range expression is "from" the smaller node number "to" the larger node number, and this form implies that the range in the opposite direction is the same (as one would normally expect). A range expression with those values reversed is interpreted as an overriding range, indicating that the distance between the two nodes is not symmetrical – weird, but for some environments we have to be able to say this because the forward and reverse data paths are very different (e.g., configured to go through different antenna complexes).
- Be very careful when you code Internet-style (TCP, UDP, etc.) duct names. These names have to be correct and consistent, or else you will see no flow of data. Don't ever use "127.0.0.1" as a hostname unless you are certain you will never want to communicate with nodes on any other

machines. If your hostname is mapped to an IP address in /etc/hosts rather than by DNS, make sure that the address that the hostname maps to is not 127.0.0.1, for the same reason.