JPL D-48259

Interplanetary Overlay Network (ION)
Design and Operation

V3.1

28 September 2012

Acknowledgment

The very substantial contributions of the ION support staff at Ohio University – and especially David Young – to the documentation of the ION system are gratefully acknowledged.

Document Owner:

_____ _____
Scott Burleigh Date
DINET Cognizant Engineer for Flight Software

Approved by:

_____ _____
Ross Jones Date
DINET Project Manager

Approved by:

_____ _____
André Girerd Date
DINET System Engineer

Approved by:

_____ _____
Margaret Lam Date
DINET Software Quality Assurance Engineer

Prepared by:

_____ _____
Scott Burleigh Date
DINET Cognizant Engineer for Flight Software

Concurred by:

_____ _____
Son Ho Date
DINET Cognizant Engineer for Ground Data System

Concurred by:

_____ _____
Leigh Torgerson Date
DINET Cognizant Engineer for Experiment Operation Center

DOCUMENT CHANGE LOG

Change Number	Change Date	Pages Affected	Changes/ Notes	General Comments
V3.1	9/28/2012		Document ION 3.1 release features.	
V3.0	3/22/2012		Align with ION 3.0 release	
V1.13	10/13/2011		Updates for SourceForge release	
V1.12	6/11/2010		Updates for second open source release (2.2)	
V1.11	12/11/2009		BRS updates, multi-node config.	
V1.10	10/23/2009		Final additions prior to DINET 2 experiment	
V1.9	6/29/2009		Add updates for DINET 2, including CFDP and ionsec.	
V1.8	2/6/2009		Update discussion of Contact Graph Routing; document status msg formats.	
V1.7	12/1/2008		Add documentation for one-way-light-time simulators, BP extension interface.	
V1.6	10/03/2008		Add documentation of sm_SemUnend.	
V1.5	09/20/2008		Revisions requested by JPL SQA.	
V1.4	07/31/2008		Add a section on optimizing an ION-based network; tuning.	
V1.3	07/08/2008		Revised some details of Contact Graph Routing.	
V1.2	05/24/2008		Revised man pages for bptrace, ltprc, bprc.	
V1.1	05/18/2008		Some additional diagrams.	
V1.0	04/28/2008		Initial version of the ION design and operations manual.	

Contents

Figures

1 Design

The Interplanetary Overlay Network (ION) software distribution is an implementation of Delay-Tolerant Networking (DTN) architecture as described in Internet RFC 4838. It is designed to enable inexpensive insertion of DTN functionality into embedded systems such as robotic spacecraft. The intent of ION deployment in space flight mission systems is to reduce cost and risk in mission communications by simplifying the construction and operation of automated digital data communication networks spanning space links, planetary surface links, and terrestrial links.

A comprehensive overview of DTN is beyond the scope of this document. Very briefly, though, DTN is a digital communication networking technology that enables data to be conveyed between two communicating entities automatically and reliably even if one or more of the network links in the end-to-end path between those entities is subject to very long signal propagation latency and/or prolonged intervals of unavailability.

The DTN architecture is much like the architecture of the Internet, except that it is one layer higher in the familiar ISO protocol "stack". The DTN analog to the Internet Protocol (IP), called "Bundle Protocol" (BP), is designed to function as an "overlay" network protocol that interconnects "internets" – including both Internet-structured networks and also data paths that utilize only space communication links as defined by the Consultative Committee for Space Data Systems (CCSDS) – in much the same way that IP interconnects "subnets" such as those built on Ethernet, SONET, etc. By implementing the DTN architecture, ION provides communication software configured as a protocol stack that looks like this:

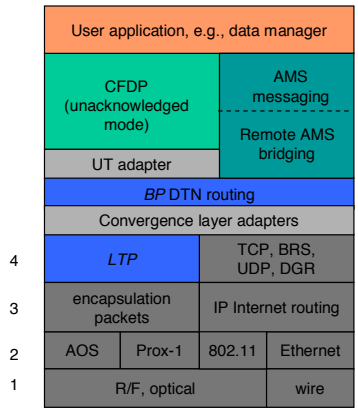


Figure 1 DTN protocol stack

Data traversing a DTN are conveyed in DTN *bundles* – which are functionally analogous to IP packets – between BP *endpoints* which are functionally analogous to sockets. Multiple BP endpoints may reside on the same computer – termed a *node* – just as multiple sockets may reside on the same computer (host or router) in the Internet.

BP endpoints are identified by Universal Record Identifiers (URIs), which are ASCII text strings of the general form:

scheme_name:*scheme_specific_part*

For example:

dtn://topquark.caltech.edu/mail

But for space flight communications this general textual representation might impose more transmission overhead than missions can afford. For this reason, ION is optimized for networks of endpoints whose IDs conform more narrowly to the following scheme:

ipn:*node_number*.*service_number*

This enables them to be abbreviated to pairs of unsigned binary integers via a technique called Compressed Bundle Header Encoding (CBHE). CBHE-conformant BP *endpoint IDs* (EIDs) are not only functionally similar to Internet socket addresses but also structurally similar: node numbers are roughly analogous to Internet node numbers (IP addresses), in that they typically identify the flight or ground data system computers on which network software executes, and service numbers are roughly analogous to TCP and UDP port numbers.

More generally, the node numbers in CBHE-conformant BP endpoint IDs are one manifestation of the fundamental ION notion of *network node number*: in the ION architecture there is a natural one-to-one mapping not only between node numbers and BP endpoint node numbers but also between node numbers and:

- LTP engine IDs
- AMS continuum numbers
- CFDP entity numbers

Starting with version 3.1 of ION, this endpoint naming rule is experimentally extended to accommodate *bundle multicast*, i.e., the delivery of copies of a single transmitted bundle to multiple nodes at which interest in that bundle's payload has been expressed. Multicast in ION – "Interplanetary Multicast" (IMC) – is accomplished by simply issuing a bundle whose destination endpoint ID conforms to the following scheme:

imc:*group_number*.*service_number*

A copy of the bundle will automatically be delivered at every node that has registered in the destination endpoint.

(Note: for now, the operational significance of a given group number must be privately negotiated among ION users. If this multicast mechanism proves useful, IANA may at some point establish a registry for IMC group numbers.)

1.1 Structure and function

The ION distribution comprises the following software packages:

- ici (Interplanetary Communication Infrastructure), a set of general-purpose libraries providing common functionality to the other packages. The ici package

includes a security policy component that supports the implementation of security mechanisms at multiple layers of the protocol stack.

- ltp (Licklider Transmission Protocol), a core DTN protocol that provides transmission reliability based on delay-tolerant acknowledgments, timeouts, and retransmissions. The LTP specification is defined in Internet RFC 5326.
- bp (Bundle Protocol), a core DTN protocol that provides delay-tolerant forwarding of data through a network in which continuous end-to-end connectivity is never assured, including support for delay-tolerant dynamic routing. The BP specification is defined in Internet RFC 5050.
- dgr (Datagram Retransmission), an alternative implementation of LTP that is designed for use in the Internet. Equipped with algorithms for TCP-like congestion control, DGR enables data to be transmitted via UDP with reliability comparable to that provided by TCP. The dgr system is provided primarily for the conveyance of Meta-AMS (see below) protocol traffic in an Internet-like environment.
- ams (Asynchronous Message Service), an application-layer service that is not part of the DTN architecture but utilizes underlying DTN protocols. AMS comprises three protocols supporting the distribution of brief messages within a network:
    - The core AAMS (Application AMS) protocol, which does message distribution on both the publish/subscribe model and the client/server model, as required by the application.
    - The MAMS (Meta-AMS) protocol, which distributes control information enabling the operation of the Application AMS protocol.
    - The RAMS (Remote AMS) protocol, which performs aggregated message distribution to end nodes that may be numerous and/or accessible only over very expensive links, using an aggregation tree structure similar to the distribution trees used by Internet multicast technologies.
- cfdp (CCSDS File Delivery Protocol), another application-layer service that is not part of the DTN architecture but utilizes underlying DTN protocols. CFDP performs the segmentation, transmission, reception, reassembly, and delivery of files in a delay-tolerant manner. ION's implementation of CFDP conforms to the "class 1" definition of the protocol in the CFDP standard, utilizing DTN (BP, nominally over LTP) as its "unitdata transport" layer.
- bss (Bundle Streaming Service), a system for efficient data streaming over a delay-tolerant network. The bss package includes (a) a forwarder that preserves in-order arrival of all data that were never lost en route, yet ensures that all data arrive at the destination eventually, and (b) a library for building delay-tolerant streaming applications, which enables low-latency presentation of streamed data received in real time while offering rewind/playback capability for the entire stream including late-arriving retransmitted data.

Taken together, the packages included in the ION software distribution constitute a communication capability characterized by the following operational features:

- Reliable conveyance of data over a delay-tolerant network (*dtnet*), i.e., a network in which it might never be possible for any node to have reliable information about the detailed current state of any other node.
- Built on this capability, reliable data streaming, reliable file delivery, and reliable distribution of short messages to multiple recipients (subscribers) residing in such a network.
- Management of traffic through such a network, taking into consideration:
    - requirements for data security
    - scheduled times and durations of communication opportunities
    - fluctuating limits on data storage and transmission resources
    - data rate asymmetry
    - the sizes of application data units
    - and user-specified final destination, priority, and useful lifetime for those data units.
- Facilities for monitoring the performance of the network.
- Robustness against node failure.
- Portability across heterogeneous computing platforms.
- High speed with low overhead.
- Easy integration with heterogeneous underlying communication infrastructure, ranging from Internet to dedicated spacecraft communication links.

1.2 Constraints on the Design

A DTN implementation intended to function in an interplanetary network environment – specifically, aboard interplanetary research spacecraft separated from Earth and one another by vast distances – must operate successfully within two general classes of design constraints: link constraints and processor constraints.

1.2.1 Link constraints

All communications among interplanetary spacecraft are, obviously, wireless. Less obviously, those wireless links are generally slow and are usually asymmetric.

The electrical power provided to on-board radios is limited and antennae are relatively small, so signals are weak. This limits the speed at which data can be transmitted intelligibly from an interplanetary spacecraft to Earth, usually to some rate on the order of 256 Kbps to 6 Mbps.

The electrical power provided to transmitters on Earth is certainly much greater, but the sensitivity of receivers on spacecraft is again constrained by limited power and antenna mass allowances. Because historically the volume of command traffic that had to be sent to spacecraft was far less than the volume of telemetry the spacecraft were expected to

return, spacecraft receivers have historically been engineered for even lower data rates from Earth to the spacecraft, on the order of 1 to 2 Kbps.

As a result, the cost per octet of data transmission or reception is high and the links are heavily subscribed. Economical use of transmission and reception opportunities is therefore important, and transmission is designed to enable useful information to be obtained from brief communication opportunities: units of transmission are typically small, and the immediate delivery of even a small part (carefully delimited) of a large data object may be preferable to deferring delivery of the entire object until all parts have been acquired.

1.2.2 Processor constraints

The computing capability aboard a robotic interplanetary spacecraft is typically quite different from that provided by an engineering workstation on Earth. In part this is due, again, to the limited available electrical power and limited mass allowance within which a flight computer must operate. But these factors are exacerbated by the often intense radiation environment of deep space. In order to minimize errors in computation and storage, flight processors must be radiation-hardened and both dynamic memory and non-volatile storage (typically flash memory) must be radiation-tolerant. The additional engineering required for these adaptations takes time and is not inexpensive, and the market for radiation-hardened spacecraft computers is relatively small; for these reasons, the latest advances in processing technology are typically not available for use on interplanetary spacecraft, so flight computers are invariably slower than their Earth-bound counterparts. As a result, the cost per processing cycle is high and processors are heavily subscribed; economical use of processing resources is very important.

The nature of interplanetary spacecraft operations imposes a further constraint. These spacecraft are wholly robotic and are far beyond the reach of mission technicians; hands-on repairs are out of the question. Therefore the processing performed by the flight computer must be highly reliable, which in turn generally means that it must be highly predictable. Flight software is typically required to meet “hard” real-time processing deadlines, for which purpose it must be run within a hard real-time operating system (RTOS).

One other implication of the requirement for high reliability in flight software is that the dynamic allocation of system memory may be prohibited except in certain well-understood states, such as at system start-up. Unrestrained dynamic allocation of system memory introduces a degree of unpredictability into the overall flight system that can threaten the reliability of the computing environment and jeopardize the health of the vehicle.

1.3 Design Principles

The design of the ION software distribution reflects several core principles that are intended to address these constraints.

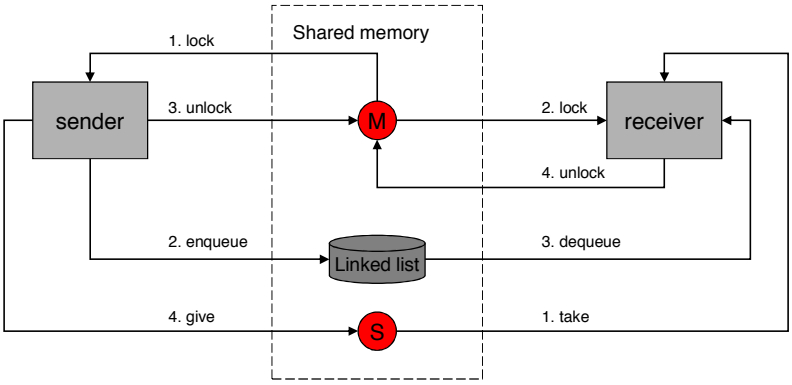


Figure 2 ION inter-task communication

1.3.1 Shared memory

Since ION must run on flight processors, it had to be designed to function successfully within an RTOS. Many real-time operating systems improve processing determinism by omitting the support for protected-memory models that is provided by Unix-like operating systems: all tasks have direct access to all regions of system memory. (In effect, all tasks operate in kernel mode rather than in user mode.) ION therefore had to be designed with no expectation of memory protection.

But universally shared access to all memory can be viewed not only as a hazard but also as an opportunity. Placing a data object in shared memory is an extremely efficient means of passing data from one software task to another.

ION is designed to exploit this opportunity as fully as possible. In particular, virtually all inter-task communication in ION follows the model shown in Figure 2:

- The sending task takes a mutual exclusion semaphore (mutex) protecting a linked list in shared memory (either DRAM or non-volatile memory), appends a data item to the list, releases the mutex, and gives a "signal" semaphore associated with the list to announce that the list is now non-empty.
- The receiving task, which is already pended on the linked list's associated signal semaphore, resumes execution when the semaphore is given. It takes the associated mutex, extracts the next data item from the list, releases the mutex, and proceeds to operate on the data item from the sending task.

Semaphore operations are typically extremely fast, as is the storage and retrieval of data in memory, so this inter-task communication model is suitably efficient for flight software.

1.3.2 Zero-copy procedures

Given ION's orientation toward the shared memory model, a further strategy for processing efficiency offers itself: if the data item appended to a linked list is merely a pointer to a large data object, rather than a copy, then we can further reduce processing overhead by eliminating the cost of byte-for-byte copying of large objects.

Moreover, in the event that multiple software elements need to access the same large object at the same time, we can provide each such software element with a pointer to the object rather than its own copy (maintaining a count of references to assure that the object is not destroyed until all elements have relinquished their pointers). This serves to reduce somewhat the amount of memory needed for ION operations.

1.3.3 Highly distributed processing

The efficiency of inter-task communications based on shared memory makes it practical to distribute ION processing among multiple relatively simple pipelined tasks rather than localize it in a single, somewhat more complex daemon. This strategy has a number of advantages:

- The simplicity of each task reduces the sizes of the software modules, making them easier to understand and maintain, and thus it can somewhat reduce the incidence of errors.
- The scope of the ION operating stack can be adjusted incrementally at run time, by spawning or terminating instances of configurable software elements, without increasing the size or complexity of any single task and without requiring that the stack as a whole be halted and restarted in a new configuration. In theory, a module could even be upgraded with new functionality and integrated into the stack without interrupting operations.
- The clear interfaces between tasks simplify the implementation of flow control measures to prevent uncontrolled resource consumption.

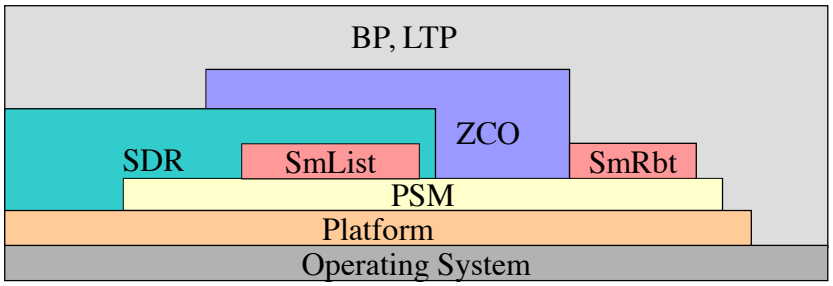
1.3.4 Portability

Designs based on these kinds of principles are foreign to many software developers, who may be far more comfortable in development environments supported by protected memory. It is typically much easier, for example, to develop software in a Linux environment than in VxWorks 5.4. However, the Linux environment is not the only one in which ION software must ultimately run.

Consequently, ION has been designed for easy portability. POSIX™ API functions are widely used, and differences in operating system support that are not concealed within the POSIX abstractions are mostly encapsulated in two small modules of platform-sensitive ION code. The bulk of the ION software runs, without any source code modification whatsoever, equally well in Linux™ (Red Hat®, Fedora™, and Ubuntu™, so far), FreeBSD®, Solaris® 9, Microsoft Windows (the MinGW environment), OS/X®, VxWorks® 5.4, and RTEMS™, on both 32-bit and 64-bit processors. Developers may compile and test ION modules in whatever environment they find most convenient.

1.4 Organizational Overview

Two broad overviews of the organization of ION may be helpful at this point. First, here is a summary view of the main functional dependencies among ION software elements:



BP, LTP	Bundle Protocol and Licklider Transmission Protocol libraries and daemons
ZCO	Zero-copy objects capability: minimize data copying up and down the stack
SDR	Spacecraft Data Recorder: persistent object database in shared memory, using PSM and SmList
SmList	linked lists in shared memory using PSM
SmRbt	red-black trees in shared memory using PSM
PSM	Personal Space Management: memory management within a pre-allocated memory partition
Platform	common access to O.S.: shared memory, system time, IPC mechanisms
Operating System	POSIX thread spawn/destroy, file system, time

Figure 3 ION software functional dependencies

That is, BP and LTP invoke functions provided by the sdr, zco, psm, and platform elements of the ici package, in addition to functions provided by the operating system itself; the zco functions themselves also invoke sdr, psm, and platform functions; and so on.

Second, here is a summary view of the main line of data flow in ION's DTN protocol implementations:

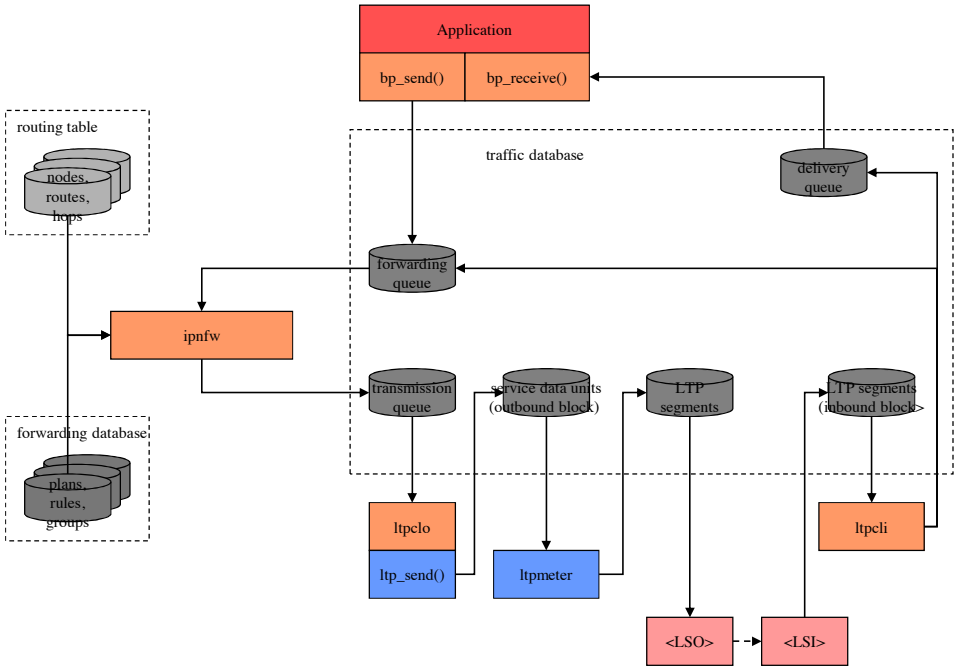


Figure 4 Main line of ION data flow

Note that data objects residing in shared memory, many of them in a nominally non-volatile SDR data store, constitute the central organizing principle of the design. Here as in other diagrams showing data flow in this document:

- Ordered collections of data objects are shown as cylinders.
- Darker greyscale data entities indicate data that are managed in the SDR data store, while lighter greyscale data entities indicate data that are managed in volatile DRAM to improve performance.
- Rectangles indicate processing elements (tasks, processes, threads), sometimes with library references specifically identified.

A few notes on this main line data flow:

- For simplicity, the data flow depicted here is a "loopback" flow in which a single BP "node" is shown sending data to itself (a useful configuration for test purposes). To depict typical operations over a network we would need two instances of this node diagram, such that the <LSO> task of one node is shown sending data to the <LSI> task of the other and vice versa.
- A BP application or application service (such as Remote AMS) that has access to the local BP node – for our purposes, the "sender" – invokes the `bp_send` function to send a unit of application data to a remote counterpart. The destination of the application data unit is expressed as a BP endpoint ID (EID). The application data unit is encapsulated in a bundle and is queued for forwarding.
- The forwarder task identified by the "scheme" portion of the bundle's destination EID removes the bundle from the forwarding queue and computes a route to the destination EID. The first node on the route, to which the local node is able to transmit data directly via some underlying "convergence layer" (CL) protocol, is termed the "proximate node" for the computed route. The forwarder appends the bundle to one of the transmission queues for the CL-protocol-specific interface to the proximate node, termed an *outduct*. Each outduct is serviced by some CL-specific output task that communicates with the proximate node – in this case, the LTP output task **ltpclo**. (Other CL protocols supported by ION include TCP and UDP.)
- The output task for LTP transmission to the selected proximate node removes the bundle from the transmission queue and invokes the `ltp_send` function to append it to a *block* that is being assembled for transmission to the proximate node. (Because LTP acknowledgment traffic is issued on a per-block basis, we can limit the amount of acknowledgment traffic on the network by aggregating multiple bundles into a single block rather than transmitting each bundle in its own block.)
- The **ltpmeter** task for the selected proximate node divides the aggregated block into multiple segments and enqueues them for transmission by underlying link-layer transmission software, such as an implementation of the CCSDS AOS protocol.

- Underlying link-layer software at the sending node transmits the segments to its counterpart at the proximate node (the receiver), where they are used to reassemble the transmission block.
- The receiving node's input task for LTP reception extracts the bundles from the reassembled block and dispatches them: each bundle whose final destination is some other node is queued for forwarding, just like bundles created by local applications, while each bundle whose final destination is the local node is queued for delivery to whatever application "opens" the BP endpoint identified by the bundle's final destination endpoint ID. (Note that a multicast bundle may be both queued for forwarding, possibly to multiple neighboring nodes, and also queued for delivery.)
- The destination application or application service at the receiving node opens the appropriate BP endpoint and invokes the `bp_receive` function to remove the bundle from the associated delivery queue and extract the original application data unit, which it can then process.

Finally, note that the data flow shown here represents the sustained operational configuration of a node that has been successfully instantiated on a suitable computer. The sequence of operations performed to reach this configuration is not shown. That startup sequence will necessarily vary depending on the nature of the computing platform and the supporting link services. Broadly, the first step normally is to run the **ionadmin** utility program to initialize the data management infrastructure required by all elements of ION. Following this initialization, the next steps normally are (a) any necessary initialization of link service protocols, (b) any necessary initialization of convergence-layer protocols (e.g., LTP – the **ltpadmin** utility program), and finally (c) initialization of the Bundle Protocol by means of the **bpadmin** utility program. BP applications should not try to commence operation until BP has been initialized.

1.5 Resource Management in ION

Successful Delay-Tolerant Networking relies on retention of bundle protocol agent state information – including protocol traffic that is awaiting a transmission opportunity – for potentially lengthy intervals. The nature of that state information will fluctuate rapidly as the protocol agent passes through different phases of operation, so efficient management of the storage resources allocated to state information is a key consideration in the design of ION.

Two general classes of storage resources are managed by ION: volatile "working memory" and non-volatile "heap".

1.5.1 Working Memory

ION's "working memory" is a fixed-size pool of shared memory (dynamic RAM) that is allocated from system RAM at the time the bundle protocol agent commences operation. Working memory is used by ION tasks to store temporary data of all kinds: linked lists, red-black trees, transient buffers, volatile databases, etc. All intermediate data products and temporary data structures that ought not to be retained in the event of a system power cycle are written to working memory.

Data structures residing in working memory may be shared among ION tasks or may be created and managed privately by individual ION tasks. The dynamic allocation of working memory to ION tasks is accomplished by the Personal Space Management (PSM) service, described later. All of the working memory for any single ION bundle protocol agent is managed as a single PSM "partition". The size of the partition is specified in the **wmSize** parameter of the ionconfig file supplied at the time ION is initialized.

1.5.2 Heap

ION's "heap" is a fixed-size pool of notionally non-volatile storage that is likewise allocated at the time the bundle protocol agent commences operation. This notionally non-volatile space **may** occupy a fixed-size pool of shared memory (dynamic RAM, which might or might not be battery-backed), or it **may** occupy only a single fixed-size file in the file system, or it may occupy both. In the latter case, all heap data are written both to memory and to the file but are read only from memory; this configuration offers the reliable non-volatility of file storage coupled with the high performance of retrieval from dynamic RAM.

We characterize ION's heap storage as "notionally" non-volatile because the heap may be configured to reside only in memory (or, for that matter, in a file that resides in the file system of a RAM disk). When the heap resides only in memory, its contents are truly non-volatile only if that memory is battery-backed. Otherwise heap storage is in reality as volatile as working memory: heap contents will be lost upon a system power cycle (which may in fact be the preferred behavior for any given deployment of ION). However, the heap should not be thought of as "memory" even when it in fact resides only in DRAM, just as a disk device should not be thought of as "memory" even when it is in fact a ram disk.

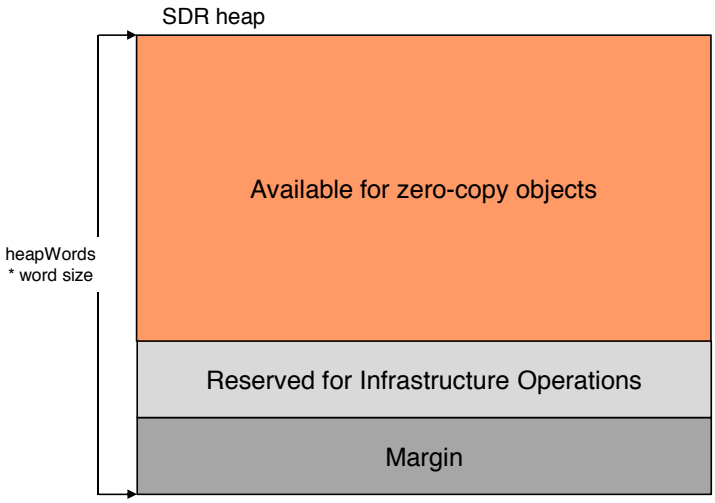


Figure 5 ION heap space use

The ION heap is used for storage of data that (in at least some deployments) would have to be retained in the event of a system power cycle to ensure the correct continued operation of the node. For example, all queues of bundles awaiting route computation,

transmission, or delivery reside in the node's heap. So do the non-volatile databases for all of the protocols implemented within ION, together with all of the node's persistent configuration parameters.

The dynamic allocation of heap space to ION tasks is accomplished by the Simple Data Recorder (SDR) service, described later. The entire heap for any single ION bundle protocol agent is managed as a single SDR "data store".

Space within the ION heap is apportioned as shown in **Error! Reference source not found.**. The total number of bytes of storage space in the heap is computed as the product of the size of a "word" on the deployment platform (normally the size of a pointer) multiplied by the value of the **heapWords** parameter of the ionconfig file supplied at the time ION is initialized. Of this total, 20% is normally reserved as margin and another 20% is normally reserved for various infrastructure operations. (Both of these percentages are macros that may be overridden at compile time.) The remainder is available for storage of protocol state data in the form of "zero-copy objects", described later. At any given moment, the data encapsulated in a zero-copy object may "belong" to any one of the protocols in the ION stack (AMS, CFDP, BP, LTP), depending on processing state; the available heap space is a single common resource to which all of the protocols share concurrent access.

Because the heap is used to store queues of bundles awaiting processing, blocks of LTP data awaiting transmission or reassembly, etc., the heap for any single ION node must be large enough to contain the maximum volume of such data that the node will be required to retain during operations. Demand for heap space is substantially mitigated if most of the application data units passed to ION for transmission are file-resident, as the file contents themselves need not be copied into the heap. In general, however, computing the optimum ION heap size for a given deployment remains a research topic.

1.6 Package Overviews

1.6.1 Interplanetary Communication Infrastructure (ICI)

The ICI package in ION provides a number of core services that, from ION's point of view, implement what amounts to an extended POSIX-based operating system. ICI services include the following:

1. Platform

The platform system contains operating-system-sensitive code that enables ICI to present a single, consistent programming interface to those common operating system services that multiple ION modules utilize. For example, the platform system implements a standard semaphore abstraction that may invisibly be mapped to underlying POSIX semaphores, SVR4 IPC semaphores, Windows Events, or VxWorks semaphores, depending on which operating system the package is compiled for. The platform system also implements a standard shared-memory abstraction, enabling software running on operating systems both with and without memory protection to participate readily in ION's shared-memory-based computing environment.

2. Personal Space Management (PSM)

Although sound flight software design may prohibit the uncontrolled dynamic management of system memory, private management of assigned, fixed blocks of system memory is standard practice. Often that private management amounts to merely controlling the reuse of fixed-size rows in static tables, but such techniques can be awkward and may not make the most efficient use of available memory. The ICI package provides an alternative, called PSM, which performs high-speed dynamic allocation and recovery of variable-size memory objects within an assigned memory block of fixed size. A given PSM-managed memory block may be either private or shared memory.

3. Memmgr

The static allocation of privately-managed blocks of system memory for different purposes implies the need for multiple memory management regimes, and in some cases a program that interacts with multiple software elements may need to participate in the private shared-memory management regimes of each. ICI's memmgr system enables multiple memory managers – for multiple privately-managed blocks of system memory – to coexist within ION and be concurrently available to ION software elements.

4. Lyst

The lyst system is a comprehensive, powerful, and efficient system for managing doubly-linked lists in private memory. It is the model for a number of other list management systems supported by ICI; as noted earlier, linked lists are heavily used in ION inter-task communication.

5. Llcv

The llcv (Linked-List Condition Variables) system is an inter-thread communication abstraction that integrates POSIX thread condition variables (vice semaphores) with doubly-linked lists in private memory.

6. Smlist

Smlist is another doubly-linked list management service. It differs from lyst in that the lists it manages reside in shared (rather than private) DRAM, so operations on them must be semaphore-protected to prevent race conditions.

7. SmRbt

The SmRbt service provides mechanisms for populating and navigating "red/black trees" (RBTs) residing in shared DRAM. RBTs offer an alternative to linked lists: like linked lists they can be navigated as queues, but locating a single element of an RBT by its "key" value can be much quicker than the equivalent search through an ordered linked list.

8. Simple Data Recorder (SDR)

SDR is a system for managing non-volatile storage, built on exactly the same model as PSM. Put another way, SDR is a small and simple "persistent object" system or "object database" management system. It enables straightforward management of linked lists (and other data structures of arbitrary complexity) in non-volatile storage, notionally within a single file whose size is pre-defined and fixed.

SDR includes a transaction mechanism that protects database integrity by ensuring that the failure of any database operation will cause all other operations undertaken within the same transaction to be backed out. The intent of the system is to assure retention of coherent protocol engine state even in the event of an unplanned flight computer reboot in the midst of communication activity.

9. Sptrace

The sptrace system is an embedded diagnostic facility that monitors the performance of the PSM and SDR space management systems. It can be used, for example, to detect memory "leaks" and other memory management errors.

10. Zco

ION's zco (zero-copy objects) system leverages the SDR system's storage flexibility to enable user application data to be encapsulated in any number of layers of protocol without copying the successively augmented protocol data unit from one layer to the next. It also implements a reference counting system that enables protocol data to be processed safely by multiple software elements concurrently – e.g., a bundle may be both delivered to a local endpoint and, at the same time, queued for forwarding to another node – without requiring that distinct copies of the data be provided to each element.

11. Rfx

The ION rfx (R/F Contacts) system manages lists of scheduled communication opportunities in support of a number of LTP and BP functions.

12. Ionsec

The IONSEC (ION security) system manages information that supports the implementation of security mechanisms in the other packages: security policy rules and computation keys.

1.6.2 Licklider Transmission Protocol (LTP)

The ION implementation of LTP conforms fully to RFC 5326, but it also provides two additional features that enhance functionality without affecting interoperability with other implementations:

- The service data units – nominally bundles – passed to LTP for transmission may be aggregated into larger blocks before segmentation. By controlling block size we can control the volume of acknowledgment traffic generated as blocks are received, for improved accommodation of highly asynchronous data rates.
- The maximum number of transmission sessions that may be concurrently managed by LTP (a protocol control parameter) constitutes a transmission "window" – the basis for a delay-tolerant, non-conversational flow control service over interplanetary links.

In the ION stack, LTP serves effectively the same role that is performed by an LLC protocol (such as IEEE 802.2) in the Internet architecture, providing flow control and retransmission-based reliability between topologically adjacent bundle protocol agents.

All LTP session state is safely retained in the ION heap for rapid recovery from a spacecraft or software fault.

1.6.3 Bundle Protocol (BP)

The ION implementation of BP conforms fully to RFC 5050, including support for the following standard capabilities:

- Prioritization of data flows
- Proactive bundle fragmentation
- Bundle reassembly from fragments
- Flexible status reporting
- Custody transfer, including re-forwarding of custodial bundles upon timeout interval expiration or failure of nominally reliable convergence-layer transmission

The system also provides three additional features that enhance functionality without affecting interoperability with other implementations:

- Rate control provides support for congestion forecasting and avoidance.
- Bundle headers are encoded into compressed form (CBHE, as noted earlier) before issuance, to reduce protocol overhead and improve link utilization.
- Bundles may be "multicast" to all nodes that have registered within a given multicast group endpoint.

In addition, ION BP includes a system for computing dynamic routes through time-varying network topology assembled from scheduled, bounded communication opportunities. This system, called "Contact Graph Routing," is described later in this Guide.

In short, BP serves effectively the same role that is performed by IP in the Internet architecture, providing route computation, forwarding, congestion avoidance, and control over quality of service.

All bundle transmission state is safely retained in the ION heap for rapid recovery from a spacecraft or software fault.

1.6.4 Asynchronous Message Service (AMS)

The ION implementation of the CCSDS AMS standard conforms fully to CCSDS 735.0-B-1. AMS is a data system communications architecture under which the modules of mission systems may be designed as if they were to operate in isolation, each one producing and consuming mission information without explicit awareness of which other modules are currently operating. Communication relationships among such modules are self-configuring; this tends to minimize complexity in the development and operations of modular data systems.

A system built on this model is a "society" of generally autonomous inter-operating modules that may fluctuate freely over time in response to changing mission objectives, modules' functional upgrades, and recovery from individual module failure. The purpose

of AMS, then, is to reduce mission cost and risk by providing standard, reusable infrastructure for the exchange of information among data system modules in a manner that is simple to use, highly automated, flexible, robust, scalable, and efficient.

A detailed discussion of AMS is beyond the scope of this Design Guide. For more information, please see the AMS Programmer's Guide.

1.6.5 Datagram Retransmission (DGR)

The DGR package in ION is an alternative implementation of LTP that is designed to operate responsibly – i.e., with built-in congestion control – in the Internet or other IP-based networks. It is provided as a candidate "primary transfer service" in support of AMS operations in an Internet-like (non-delay-tolerant) environment. The DGR design combines LTP's concept of concurrent transmission transactions with congestion control and timeout interval computation algorithms adapted from TCP.

1.6.6 CCSDS File Delivery Protocol (CFDP)

The ION implementation of CFDP conforms fully to Service Class 1 (Unreliable Transfer) of CCSDS 727.0-B-4, including support for the following standard capabilities:

- Segmentation of files on user-specified record boundaries.
- Transmission of file segments in protocol data units that are conveyed by an underlying Unitdata Transfer service, in this case the DTN protocol stack. File data segments may optionally be protected by CRCs. When the DTN protocol stack is configured for reliable data delivery (i.e., with BP custody transfer running over a reliable convergence-layer protocol such as LTP), file delivery is reliable; CFDP need not perform retransmission of lost data itself.
- Reassembly of files from received segments, possibly arriving over a variety of routes through the delay-tolerant network. The integrity of the delivered files is protected by checksums.
- User-specified fault handling procedures.
- Operations (e.g., directory creation, file renaming) on remote file systems.

All CFDP transaction state is safely retained in the ION heap for rapid recovery from a spacecraft or software fault.

1.6.7 Bundle Streaming Service (BSS)

The BSS service provided in ION enables a stream of video, audio, or other continuously generated application data units, transmitted over a delay-tolerant network, to be presented to a destination application in two useful modes concurrently:

- In the order in which the data units were generated, with the least possible end-to-end delivery latency, but possibly with some gaps due to transient data loss or corruption.

- In the order in which the data units were generated, without gaps (i.e., including lost or corrupt data units which were omitted from the real-time presentation but were subsequently retransmitted), but in a non-real-time "playback" mode.

1.7 Acronyms

BP	Bundle Protocol
BSP	Bundle Security Protocol
BSS	Bundle Streaming Service
CCSDS	Consultative Committee for Space Data Systems
CFDP	CCSDS File Delivery Protocol
CGR	Contact Graph Routing
CL	convergence layer
CLI	convergence layer input
CLO	convergence layer output
DTN	Delay-Tolerant Networking
ICI	Interplanetary Communication Infrastructure
ION	Interplanetary Overlay Network
LSI	link service input
LSO	link service output
LTP	Licklider Transmission Protocol
OWLT	one-way light time
RFC	request for comments
RFX	Radio (R/F) Contacts
RTT	round-trip time
TTL	time to live

1.8 Network Operation Concepts

A small number of network operation design elements – fragmentation and reassembly, bandwidth management, and delivery assurance (retransmission) – can potentially be addressed at multiple layers of the protocol stack, possibly in different ways for different reasons. In stack design it's important to allocate this functionality carefully so that the effects at lower layers complement, rather than subvert, the effects imposed at higher layers of the stack. This allocation of functionality is discussed below, together with a discussion of several related key concepts in the ION design.

1.8.1 Fragmentation and Reassembly

To minimize transmission overhead and accommodate asymmetric links (i.e., limited "uplink" data rate from a ground data system to a spacecraft) in an interplanetary network, we ideally want to send "downlink" data in the largest possible aggregations – coarse-grained transmission.

But to minimize head-of-line blocking (i.e., delay in transmission of a newly presented high-priority item) and minimize data delivery latency by using parallel paths (i.e., to provide fine-grained partial data delivery, and to minimize the impact of unexpected link termination), we want to send "downlink" data in the smallest possible aggregations – fine-grained transmission.

We reconcile these impulses by doing both, but at different layers of the ION protocol stack.

First, at the application service layer (AMS and CFDP) we present relatively small application data units (ADUs) – on the order of 64 KB – to BP for encapsulation in bundles. This establishes an upper bound on head-of-line blocking when bundles are de-queued for transmission, and it provides perforations in the data stream at which forwarding can readily be switched from one link (route) to another, enabling partial data delivery at relatively fine, application-appropriate granularity.

(Alternatively, large application data units may be presented to BP and the resulting large bundles may be proactively fragmented at the time they are presented to the convergence-layer adapter. This capability is meant to accommodate environments in which the convergence-layer adapter has better information than the application as to the optimal bundle size, such as when the residual capacity of a contact is known to be less than the size of the bundle.)

Then, at the BP/LTP convergence layer adapter lower in the stack, we aggregate these small bundles into *blocks* for presentation to LTP:

> Any continuous sequence of bundles that are to be shipped to the same LTP engine and all require assured delivery may be aggregated into a single block, to reduce overhead and minimize report traffic.
>
> However, this aggregation is constrained by a block size limit rule: each block must contain an integral number N – where N is greater than zero – complete bundles, but N can only exceed 1 when the sum of the sizes of all N bundles does not exceed the *nominal block size* declared for the applicable *span* (the relationship between the local node and the receiving LTP engine) during LTP protocol configuration via **ltpadmin**.

Given a preferred block acknowledgment period – e.g., an acknowledgment traffic limit of one report per second – nominal block size is notionally computed as the amount of data that can be sent over the link to the receiving LTP engine in a single block acknowledgment period at the planned outbound data rate to that engine.

Taken together, application-level fragmentation (or BP proactive fragmentation) and LTP aggregation place an upper limit on the amount of data that would need to be re-transmitted over a given link at next contact in the event of an unexpected link

termination that caused delivery of an entire block to fail. For example, if the data rate is 1 Mbps and the nominal block size is 128 KB (equivalent to 1 second of transmission time), we would prefer to avoid the risk of having wasted five minutes of downlink in sending a 37.5 MB file that fails on transmission of the last kilobyte, forcing retransmission of the entire 37.5 MB. We therefore divide the file into, say, 1200 bundles of 32 KB each which are aggregated into blocks of 128 KB each: only a single block failed, so only that block (containing just 4 bundles) needs to be retransmitted. The cost of this retransmission is only 1 second of link time rather than 5 minutes. By controlling the cost of convergence-layer protocol failure in this way, we avoid the overhead and complexity of "reactive fragmentation" in the BP implementation.

Finally, within LTP itself we fragment the block as necessary to accommodate the Maximum Transfer Unit (MTU) size of the underlying link service, typically the transfer frame size of the applicable CCSDS link protocol.

1.8.2 Bandwidth Management

The allocation of bandwidth (transmission opportunity) to application data is requested by the application task that's passing data to DTN, but it is necessarily accomplished only at the lowest layer of the stack at which bandwidth allocation decisions can be made – and then always in the context of node policy decisions that have global effect.

The "outduct" interface to a given neighbor in the network is actually three queues of outbound bundles rather than one: one queue for each of the defined levels of priority ("class of service") supported by BP. When an application presents an ADU to BP for encapsulation in a bundle, it indicates its own assessment of the ADU's priority. Upon selection of a proximate forwarding destination node for that bundle, the bundle is appended to whichever of the neighbor interface queues corresponds to the ADU's priority.

Normally the convergence-layer output (CLO) task servicing a given outduct – e.g., the LTP output task **ltpclo** – extracts bundles in strict priority order from the heads of the outduct's three queues. That is, the bundle at the head of the highest-priority non-empty queue is always extracted.

However, if the ION_BANDWIDTH_RESERVED compiler option is selected at the time ION is built, the convergence-layer output (CLO) task servicing a given outduct extracts bundles in interleaved fashion from the heads of the outduct's three queues:

- Whenever the priority-2 ("express") queue is non-empty, the bundle at the head of that queue is the next one extracted.
- At all other times, bundles from both the priority-1 queue and the priority-0 queue are extracted, but over a given period of time twice as many bytes of priority-1 bundles will be extracted as bytes of priority-0 bundles.

CLO tasks other than **ltpclo** simply segment the extracted bundles as necessary and transmit them using the underlying convergence-layer protocol. In the case of **ltpclo**, the output task aggregates the extracted bundles into blocks as described earlier and a second daemon task named **ltpmeter** waits for aggregated blocks to be completed; **ltpmeter**, rather than the CLO task itself, segments each completed block as necessary and passes

the segments to the link service protocol that underlies LTP. Either way, the transmission ordering requested by application tasks is preserved.

1.8.3 Contact Plans

In the Internet, protocol operations can be largely driven by currently effective information that is discovered opportunistically and immediately, at the time it is needed, because the latency in communicating this information over the network is negligible: distances between communicating entities are small and connectivity is continuous. In a DTN-based network, however, ad-hoc information discovery would in many cases take so much time that it could not be completed before the information lost currency and effectiveness. Instead, protocol operations must be largely driven by information that is pre-placed at the network nodes and tagged with the dates and times at which it becomes effective. This information takes the form of *contact plans* that are managed by the R/F Contacts (rfx) services of ION's ici package.

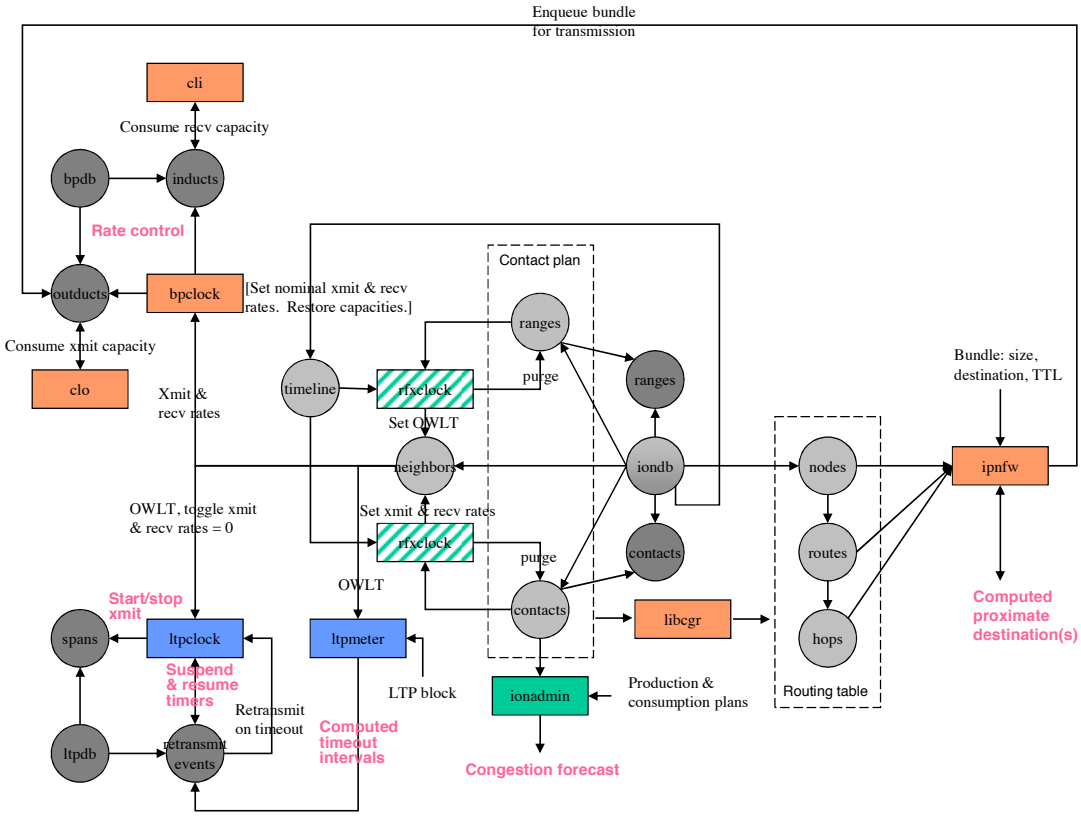


Figure 6 RFX services in ION

The structure of ION's RFX (contact plan) database, the rfx system elements that populate and use that data, and affected portions of the BP and LTP protocol state databases are shown in **Error! Reference source not found.**. (For additional details of BP and LTP database management, see the BP/LTP discussion later in this document.)

To clarify the notation of this diagram, which is also used in other database structure diagrams in this document:

- Data objects of defined structure are shown as circles. Dark greyscale indicates notionally non-volatile data retained in "heap" storage, while lighter greyscale indicates volatile data retained in dynamic random access memory.
- Solid arrows connecting circles indicate one-to-many cardinality.
- A dashed arrow between circles indicates a potentially many-to-one reference mapping.
- Arrows from processing elements (rectangles) to data entities indicate data production, while arrows from data entities to processing elements indicate data retrieval.

A *contact* is here defined as an interval during which it is expected that data will be transmitted by DTN node A (the contact's transmitting node) and most or all of the transmitted data will be received by node B (the contact's receiving node). Implicitly, the transmitting mode will utilize some "convergence-layer" protocol underneath the Bundle Protocol to effect this direct transmission of data to the receiving node. Each contact is characterized by its start time, its end time, the identities of the transmitting and receiving nodes, and the rate at which data are expected to be transmitted by the transmitting node throughout the indicated time period.

(Note that a contact is specifically ***not*** an episode of activity on a link. Episodes of activity on different links – e.g., different radio transponders operating on the same spacecraft – may well overlap, but contacts by definition cannot; they are bounded time intervals and as such are innately "tiled". For example, suppose transmission on link X from node A to node B, at data rate RX, begins at time T1 and ends at time T2; also, transmission on link Y from node A to node B, at data rate RY begins at time T3 and ends at time T4. If T1 = T3 and T2 = T4, then there is a single contact from time T1 to time T2 at data rate RX + RY. If T1 < T3 and T2 = T4, then there are two contiguous contacts: one from T1 to T3 at data rate RX, then one from T3 to T2 at data rate RX + RY. If T1 < T3 and T3<T2 < T4, then there are three contiguous contacts: one from T1 to T3 at data rate RX, then one from T3 to T2 at data rate RX + RY, then one from T2 to T4 at data rate RY. And so on.)

A *range interval* is a period of time during which the displacement between two nodes A and B is expected to vary by less than 1 light second from a stated anticipated distance. (We expect this information to be readily computable from the known orbital elements of all nodes.) Each range interval is characterized by its start time, its end time, the identities of the two nodes to which it pertains, and the anticipated approximate distance between those nodes throughout the indicated time period, to the nearest light second.

The *topology timeline* at each node in the network is a time-ordered list of scheduled or anticipated changes in the topology of the network. Entries in this list are of two types:

- Contact entries characterize scheduled contacts.
- Range entries characterize anticipated range intervals.

Each node to which, according to the RFX database, the local node transmits data directly via some convergence-layer protocol at some time is termed a *neighbor* of the local node. Each neighbor is associated with an outduct – a set of outbound transmission queues –

for the applicable BP convergence-layer (CL) protocol adapter, so bundles that are to be transmitted directly to this neighbor can simply be queued for transmission via that CL protocol (as discussed in the Bandwidth Management notes above).

At startup, and at any time while the system is running, **ionadmin** inserts and removes Contact and Range entries in the topology timeline of the RFX database. Inserting or removing a Contact or Range entry will cause routing tables to be recomputed for the destination nodes of all subsequently forwarded bundles, as described in the discussion of Contact Graph Routing below.

Once per second, the **rfxclock** task (which appears in multiple locations on the diagram to simplify the geometry) applies all topology timeline events (Contact and Range start, stop, purge) with effective time in the past. Applying a Contact event that cites a neighboring node revises the transmission or reception data rate between the local node and that Neighbor. Applying a Range event that cites a neighboring node revises the OWLT between the local node and that neighbor. Setting data rate or OWLT for a node with which the local node will at some time be in direct communication may entail creation of a Neighbor object.

1.8.4 Route Computation

ION's computation of a route for a given bundle with a given destination endpoint is accomplished by one of several methods, depending on the destination. In every case, the result of successful routing is the insertion of the bundle into an outbound transmission queue (selected according to the bundle's priority) for one or more neighboring nodes.

But before discussing these methods it will be helpful to establish some terminology:

Egress plans

ION can only forward bundles to a neighboring node by queuing them on some explicitly specified outduct. Specifications that associate neighboring nodes with outducts – possibly varying depending on the node numbers and/or service numbers of bundles' source entity IDs – are termed *egress plans*. They are retained in ION's unicast forwarding database.

Static routes

ION can be configured to forward to some specified node all bundles that are destined for a given node to which no *dynamic route* can be discovered from an examination of the contact graph, as described later. Static routing is implemented by means of the "group" mechanism described below.

Unicast

When the destination of a bundle is a single node that is registered within a known "singleton endpoint" (that is, an endpoint that is known to have exactly one member), then transmission of that bundle is termed *unicast*. For this purpose, the destination endpoint ID must be a URI formed in either the "dtn" scheme (e.g., `dtn://bobsmac/mail`) or the "ipn" scheme (e.g., `ipn:913.11`).

Unicast Groups

When unicast routes must be computed to nodes for which no contact plan information is known (e.g., the size of the network makes it impractical to distribute all Contact and Range information for all nodes to every node, or the destination nodes don't participate in Contact Graph Routing at all), the job of computing routes to all nodes may be partitioned among multiple *gateway* nodes. Each gateway is responsible for managing routing information (for example, a comprehensive contact graph) for some subset of the total network population – a *group*, comprising all nodes whose node numbers fall within the range of node numbers assigned to the gateway. A bundle destined for a node for which no dynamic route can be computed from the local node's contact graph may be routed to the gateway node for the group within whose range the destination's node number falls. Unicast groups are retained in ION's unicast forwarding database. (Note that the group mechanism implements *static routes* in CGR in addition to improving scalability.)

Multicast

When the destination of a bundle is all nodes that are registered within a known "multicast endpoint" (that is, an endpoint that is not known to have exactly one member), then transmission of that bundle is termed *multicast*. For this purpose (in ION), the destination endpoint ID must be a URI formed in the "imc" scheme (e.g., `imc:913.11`).

Multicast Groups

A *multicast group* is the set of all nodes in the network that are members of a given multicast endpoint. Forwarding a bundle to all members of its destination multicast endpoint is the responsibility of all of the multicast-aware nodes of the network. These nodes are additionally configured to be nodes of a single multicast spanning tree overlaid onto the dtnet. A single multicast tree serves to forward bundles to all multicast groups: each node of the tree manages petitions indicating which of its "relatives" (parent and children) are currently interested in bundles destined for each multicast endpoint, either natively (due to membership in the indicated group) or on behalf of more distant relatives.

1.8.4.1 Unicast

We begin unicast route computation by attempting to compute a dynamic route to the bundle's final destination node. The details of this algorithm are described in the section on ***Contact Graph Routing***, below.

If no dynamic route can be computed, but the final destination node is a "neighboring" node that is directly reachable, then we assume that taking this direct route is the best strategy unless the outduct to that neighbor is flagged as "blocked" due to a lapse in convergence-layer functionality .

Otherwise we must look for a static route. If the bundle's destination node number is in the range of node numbers assigned to the gateways for one or more groups, then we forward the bundle to that gateway node for the smallest such group. (If the gateway node is a neighbor and the outduct to that neighbor is not blocked, we simply queue the

bundle on that outduct; otherwise we similarly look up the static route for the gateway until eventually we resolve to some egress plan.)

If we can determine neither a dynamic route nor a static route for this bundle, but the reason for this failure was outduct blockage that might be resolved in the future, then the bundle is placed in a "limbo" list for future re-forwarding when some outduct is "unblocked."

Otherwise, the bundle cannot be forwarded. If custody transfer is requested for the bundle, we send a custody refusal to the bundle's current custodian; in any case, we discard the bundle.

1.8.4.2 Multicast

Multicast route computation is much simpler.

- The topology of the single network-wide multicast distribution tree is established in advance by invoking tree management library functions that declare the children and parents of each node. These functions are currently invoked only from the **imcadmin** utility program. (Manual configuration of the multicast tree seems manageable for very small and generally static networks, such as the space flight operations networks we'll be seeing over the next few years, but eventually an automated tree management protocol will be required.) Each relative of each node in the tree must also be a neighbor in the underlying dtnet: multicast routing loops are avoided at each node by forwarding each bundle only to relatives other than the one from which the bundle was received, and currently the only mechanism in ION for determining the node from which a bundle was received is to match the sender's convergence-layer endpoint ID to a plan in the unicast forwarding database – i.e., to a neighbor.
- When an endpoint for the "imc" scheme is added on an ION node – that is, when the node joins that multicast endpoint – BP administrative records noting the node's new interest in the application topic corresponding to the endpoint's group number are passed to all of the node's immediate relatives in the multicast tree. On receipt of such a record, each relative notes the sending relative's interest and forwards the record to all of its immediate relatives other than the one from which the record was received, and so on. (Deletion of endpoints results in similar propagation of cancelling administrative records.)
- A bundle whose destination endpoint cites a multicast group, whether locally sourced or received from another node:
    - Is delivered immediately, if the local node is a member of the indicated endpoint.
    - Is queued for direct transmission to every immediate relative in the multicast tree other than the one from which the bundle was received (if any).

1.8.5 Delivery Assurance

End-to-end delivery of data can fail in many ways, at different layers of the stack. When delivery fails, we can either accept the communication failure or retransmit the data structure that was transmitted at the stack layer at which the failure was detected. ION is designed to enable retransmission at multiple layers of the stack, depending on the preference of the end user application.

At the lowest stack layer that is visible to ION, the convergence-layer protocol, failure to deliver one or more segments due to segment loss or corruption will trigger segment retransmission if a "reliable" convergence-layer protocol is in use: LTP "red-part" transmission or TCP (including Bundle Relay Service, which is based on TCP)[1].

Segment loss may be detected and signaled via NAK by the receiving entity, or it may only be detected at the sending entity by expiration of a timer prior to reception of an ACK. Timer interval computation is well understood in a TCP environment, but it can be a difficult problem in an environment of scheduled contacts as served by LTP. The round-trip time for an acknowledgment dialogue may be simply twice the one-way light time (OWLT) between sender and receiver at one moment, but it may be hours or days longer at the next moment due to cessation of scheduled contact until a future contact opportunity. To account for this timer interval variability in retransmission, the **ltpclock** task infers the initiation and cessation of LTP transmission, to and from the local node, from changes in the current xmit and recv data rates in the corresponding Neighbor objects. This controls the dequeuing of LTP segments for transmission by underlying link service adapter(s) and it also controls suspension and resumption of timers, removing the effects of contact interruption from the retransmission regime. For a further discussion of this mechanism, see the section below on ***LTP Timeout Intervals***.

Note that the current OWLT in Neighbor objects is also used in the computation of the nominal expiration times of timers and that **ltpclock** is additionally the agent for LTP segment retransmission based on timer expiration.

It is, of course, possible for the nominally reliable convergence-layer protocol to fail altogether: a TCP connection might be abruptly terminated, or an LTP transmission might be canceled due to excessive retransmission activity (again possibly due to an unexpected loss of connectivity). In this event, BP itself detects the CL protocol failure and re-forwards all bundles whose acquisition by the receiving entity is presumed to have been aborted by the failure. This re-forwarding is initiated in different ways for different CL protocols, as implemented in the CL input and output adapter tasks. If immediate re-forwarding is impossible because all potentially usable outducts are blocked, the affected bundles are placed in the limbo list for future re-forwarding when some outduct is unblocked.

In addition to the implicit forwarding failure detected when a CL protocol fails, the forwarding of a bundle may be explicitly refused by the receiving entity, provided the

[1] In ION, reliable convergence-layer protocols (where available) are by default used for every bundle. The application can instead mandate selection of "best-effort" service at the convergence layer by setting the BP_BEST_EFFORT flag in the "extended class of service flags" parameter, but this feature is an ION extension that is not supported by other BP implementations at the time of this writing.

bundle is flagged for custody transfer service. A receiving node's refusal to take custody of a bundle may have any of a variety of causes: typically the receiving node either (a) has insufficient resources to store and forward the bundle, (b) has no route to the destination, or (c) will have no contact with the next hop on the route before the bundle's TTL has expired. In any case, a "custody refusal signal" (packaged in a bundle) is sent back to the sending node, which must re-forward the bundle in hopes of finding a more suitable route.

Alternatively, failure to receive a custody acceptance signal within some convergence-layer-specified or application-specified time interval may also be taken as an implicit indication of forwarding failure. Here again, when BP detects such a failure it attempts to re-forward the affected bundle, placing the bundle in the limbo list if re-forwarding is currently impossible.

In the worst case, the combined efforts of all the retransmission mechanisms in ION are not enough to ensure delivery of a given bundle, even when custody transfer is requested. In that event, the bundle's "time to live" will eventually expire while the bundle is still in custody at some node: the **bpclock** task will send a bundle status report to the bundle's report-to endpoint, noting the TTL expiration, and destroy the bundle. The report-to endpoint, upon receiving this report, may be able to initiate application-layer retransmission of the original application data unit in some way. This final retransmission mechanism is wholly application-specific, however.

1.8.6 Rate Control

In the Internet, the rate of transmission at a node can be dynamically negotiated in response to changes in level of activity on the link, to minimize congestion. On deep space links, signal propagation delays (distances) may be too great to enable effective dynamic negotiation of transmission rates. Fortunately, deep space links are operationally reserved for use by designated pairs of communicating entities over pre-planned periods of time at pre-planned rates. Provided there is no congestion inherent in the contact plan, congestion in the network can be avoided merely by adhering to the planned contact periods and data rates. *Rate control* in ION serves this purpose.

While the system is running, transmission and reception of bundles is constrained by the *current capacity* in the *throttle* of each outduct and induct. Completed bundle transmission or reception activity reduces the current capacity of the applicable duct by the capacity consumption computed for that bundle. This reduction may cause the duct's current capacity to become negative. Once the current capacity of the applicable duct's throttle goes negative, activity is blocked until non-negative capacity has been restored by **bpclock**.

Once per second, the **bpclock** task increases the current capacity of each induct and outduct throttle by one second's worth of traffic at the nominal data rate for that duct, thus enabling some possibly blocked bundle transmission and reception to proceed.

The nominal data rate for any duct of any CL protocol other than LTP (e.g., TCP) is a constant, established at the time the protocol was declared during ION initialization. For LTP, however, **bpclock** revises all ducts' nominal data rates once per second in accord with the current data rates in the corresponding Neighbor objects, as adjusted by **rfxclock**

per the contact plan. This contact-plan-based adjustment is currently not possible for CL protocols other than LTP because at present there is no straightforward mechanism for mapping from Neighbor node number to protocol duct ID for any CL protocol other than LTP. So data flow over LTP links may be episodic, but data flow over non-LTP links is always continuous.

Note that this means that:

- ION's rate control system will enable data flow over non-LTP links even if there are no contacts in the contact plan that announce it. In this context the contact plan serves only to support route computation, and no contact plan is needed at all if static routes are provided for all destinations.
- ION's rate control system will enable data flow over LTP links ***only*** if there are contacts in the contact plan that announce it. In this context, announced contacts are mandatory for at least all neighboring nodes that are reachable by LTP.

1.8.7 Flow Control

A further constraint on rates of data transmission in an ION-based network is LTP flow control. LTP is designed to enable multiple block transmission sessions to be in various stages of completion concurrently, to maximize link utilization: there is no requirement to wait for one session to complete before starting the next one. However, if unchecked this design principle could in theory result in the allocation of all memory in the system to incomplete LTP transmission sessions. To prevent complete storage resource exhaustion, we set a firm upper limit on the total number of outbound blocks that can be concurrently in transit at any given time. These limits are established by **ltpadmin** at node initialization time.

The maximum number of transmission sessions that may be concurrently managed by LTP therefore constitutes a transmission "window" – the basis for a delay-tolerant, non-conversational flow control service over interplanetary links. Once the maximum number of sessions are in flight, no new block transmission session can be initiated – regardless of how much outduct transmission capacity is provided by rate control – until some existing session completes or is canceled.

Note that this consideration emphasizes the importance of configuring the aggregation size limits and session count limits of spans during LTP initialization to be consistent with the maximum data rates scheduled for contacts over those spans.

1.8.8 Storage Management

Congestion in a dtnet is the imbalance between data enqueuing and dequeuing rates that results in exhaustion of queuing (storage) resources at a node, preventing continued operation of the protocols at that node.

In ION, the affected queuing resources are allocated from notionally non-volatile storage space in the SDR data store and/or file system. The design of ION is required to prevent resource exhaustion by simply refusing to enqueue additional data that would cause it.

However, a BP router's refusal to enqueue received data for forwarding could result in costly retransmission, data loss, and/or the "upstream" propagation of resource

exhaustion to other nodes. Therefore the ION design additionally attempts to prevent potential resource exhaustion by forecasting levels of queuing resource occupancy and reporting on any congestion that is predicted. Network operators, upon reviewing these forecasts, may revise contact plans to avert the anticipated resource exhaustion.

The non-volatile storage used by ION serves several purposes: it contains queues of bundles awaiting forwarding, transmission, and delivery; it contains LTP transmission and reception sessions, including the blocks of data that are being transmitted and received; it contains queues of LTP segments awaiting radiation; it may contain CFDP transactions in various stages of completion; and it contains protocol operational state information, such as configuration parameters, static routes, the contact graph, etc.

Effective utilization of non-volatile storage is a complex problem. Static pre-allocation of storage resources is in general less efficient (and also more labor-intensive to configure) than storage resource pooling and automatic, adaptive allocation: trying to predict a reasonable maximum size for every data storage structure and then rigidly enforcing that limit typically results in underutilization of storage resources and underperformance of the system as a whole. However, static pre-allocation is mandatory for safety-critical resources, where certainty of resource availability is more important than efficient resource utilization.

The tension between the two approaches is analogous to the tension between circuit switching and packet switching in a network: circuit switching results in underutilization of link resources and underperformance of the network as a whole (some peaks of activity can never be accommodated, even while some resources lie idle much of the time), but dedicated circuits are still required for some kinds of safety-critical communication.

So the ION data management design combines these two approaches (see 1.5 above for additional discussion of this topic):

- A fixed percentage of the total SDR data store heap size (by default, 20%) is statically allocated to the storage of protocol operational state information, which is critical to the operation of ION.
- Another fixed percentage of the total SDR data store heap size (by default, 20%) is statically allocated to "margin", a reserve that helps to insulate node management from errors in resource allocation estimates.
- The remainder of the heap, plus all pre-allocated file system space, is allocated to protocol traffic[2].

The maximum projected occupancy of the node is the result of computing a *congestion forecast* for the node, by adding to the current occupancy all anticipated net increases and decreases from now until some future time, termed the *horizon* for the forecast.

The forecast horizon is indefinite – that is, "forever" – unless explicitly declared by network management via the `ionadmin` utility program. The difference between the horizon and the current time is termed the *interval* of the forecast.

[2] Note that, in all occupancy figures, ION data management accounts not only for the sizes of the payloads of all queued bundles but also for the sizes of their headers.

Net occupancy increases and decreases are of four types:

1. Bundles that are originated locally by some application on the node, which are enqueued for forwarding to some other node.
2. Bundles that are received from some other node, which are enqueued either for forwarding to some other node or for local delivery to an application.
3. Bundles that are transmitted to some other node, which are dequeued from some forwarding queue.
4. Bundles that are delivered locally to an application, which are dequeued from some delivery queue.

The type-1 anticipated net increase (total data origination) is computed by multiplying the node's stated rate of local data production, as declared via an **ionadmin** command, by the interval of the forecast. Similarly, the type-4 anticipated net decrease (total data delivery) is computed by multiplying the node's stated rate of local data consumption, as declared via an **ionadmin** command, by the interval of the forecast. Net changes of types 2 and 3 are computed by multiplying inbound and outbound data rates, respectively, by the durations of all periods of planned communication contact that begin and/or end within the interval of the forecast.

Congestion forecasting is performed by the **ionwarn** utility program. **ionwarn** may be run independently at any time; in addition, the **ionadmin** utility program automatically runs **ionwarn** immediately before exiting if it executed any change in the contact plan, the forecast horizon, or the node's projected rates of local data production or consumption.

If the final result of the forecast computation – the maximum projected occupancy of the node over the forecast interval – is less than the total protocol traffic allocation, then no congestion is forecast. Otherwise, a congestion forecast status message is logged noting the time at which maximum projected occupancy is expected to equal the total protocol traffic allocation.

Congestion control in ION, then, has two components:

> First, ION's congestion detection is anticipatory (via congestion forecasting) rather than reactive as in the Internet.
>
> Anticipatory congestion detection is important because the second component – congestion mitigation – must also be anticipatory: it is the adjustment of communication contact plans by network management, via the propagation of revised schedules for future contacts.

(Congestion mitigation in an ION-based network is likely to remain mostly manual for many years to come, because communication contact planning involves much more than mathematics: science operations plans, thermal and power constraints, etc. It will, however, rely on the automated rate control features of ION, discussed above, which ensure that actual network operations conform to established contact plans.)

The stated local data production rate is also used to accomplish *admission control*: ION throttles the insertion of locally sourced bundles to an average rate that conforms to this

constraint. Note that it is possible to disable the metering of local data insertion by setting the production rate to -1 (in which case the consumption rate must also be set to -1 to avoid congestion forecasting anomalies); this is in fact the default, because it is convenient for research and performance benchmarking. Doing so, however, introduces the possibility that locally sourced bundles may occupy so much non-volatile storage that insufficient space remains for the reception of inbound bundles, resulting in bundle refusal and the potential propagation of congestion to "upstream" nodes. Accurately estimated local data production and consumption rates are important contributors to the efficient operation of the network as a whole.

1.8.9 Optimizing an ION-based network

ION is designed to deliver critical data to its final destination with as much certainty as possible (and optionally as soon as possible), but otherwise to try to maximize link utilization. The delivery of critical data is expedited by contact graph routing and bundle prioritization as described elsewhere. Optimizing link utilization, however, is a more complex problem.

If the volume of data traffic offered to the network for transmission is less than the capacity of the network, then all offered data should be successfully delivered[3]. But in that case the users of the network are paying the opportunity cost of whatever portion of the network capacity was not used.

Offering a data traffic volume that is exactly equal to the capacity of the network is in practice infeasible. TCP in the Internet can usually achieve this balance because it exercises end-to-end flow control: essentially, the original source of data is *blocked* from offering a message until notified by the final destination that transmission of this message can be accommodated given the current negotiated data rate over the end-to-end path (as determined by TCP's congestion control mechanisms). In a delay-tolerant network no such end-to-end negotiated data rate may exist, much less be knowable, so such precise control of data flow is impossible.[4]

The only alternative: the volume of traffic offered by the data source must be greater than the capacity of the network and the network must automatically discard excess traffic, shedding lower-priority data in preference to high-priority messages on the same path.

ION discards excess traffic proactively when possible and reactively when necessary.

Proactive data triage occurs when ION determines that it cannot compute a route that will deliver a given bundle to its final destination prior to expiration of the bundle's Time To Live (TTL). That is, a bundle may be discarded simply because its TTL is too short, but more commonly it will be discarded because the planned contacts to whichever neighboring node is first on the path to the destination are already fully subscribed: the queue of bundles awaiting transmission to that neighbor is already so long as to consume

[3] Barring data loss or corruption for which the various retransmission mechanisms in ION cannot compensate.

[4] Note that ION may indeed block the offering of a message to the network, but this is local admission control – assuring that the node's local buffer space for queuing outbound bundles is not oversubscribed – rather than end-to-end flow control. It is always possible for there to be ample local buffer space yet insufficient network capacity to convey the offered data to their final destination, and vice versa.

the entire capacity of all announced opportunities to transmit to it. Proactive data triage causes the bundle to be immediately destroyed as one for which there is "No known route to destination from here."

The determination of the degree to which a contact is subscribed is based not only on the aggregate size of the queued bundles but also on the estimated aggregate size of the overhead imposed by all the convergence-layer (CL) protocol data units – at all layers of the underlying stack – that encapsulate those bundles: packet headers, frame headers, etc. This means that the accuracy of this overhead estimate will affect the aggressiveness of ION's proactive data triage:

- If CL overhead is overestimated, the size of the bundle transmission backlog for planned contacts will be overstated, unnecessarily preventing the enqueuing of additional bundles – a potential under-utilization of available transmission capacity in the network.
- If CL overhead is underestimated, the size of the bundle transmission backlog for planned contacts will be understated, enabling the enqueuing of bundles whose transmission cannot in fact be accomplished by the network within the constraints of the current contact plan. This will eventually result in reactive data triage.

Essentially, all reactive data triage – the destruction of bundles due to TTL expiration prior to successful delivery to the final destination – occurs when the network conveys bundles at lower net rates than were projected during route computation. These performance shortfalls can have a variety of causes:

- As noted above, underestimating CL overhead causes CL overhead to consume a larger fraction of contact capacity than was anticipated, leaving less capacity for bundle transmission.
- Conversely, the total volume of traffic offered may have been accurately estimated but the amount of contact capacity may be less than was promised: a contact might be started late, stopped early, or omitted altogether, or the actual data rate on the link might be less than was advertised.
- Contacts may be more subtly shortened by the configuration of ION itself. If the clocks on nodes are known not to be closely synchronized then a "maximum clock error" of N seconds may be declared, causing reception episodes to be started locally N seconds earlier and stopped N seconds later than scheduled, to avoid missing some transmitted data because it arrived earlier or later than anticipated. But this mechanism also causes transmission episodes to be started N seconds later and stopped N seconds earlier than scheduled, to avoid transmitting to a neighbor before it is ready to receive data, and this contact truncation ensures transmission of fewer bundles than planned.
- Flow control within the convergence layer underlying the bundle protocol may constrain the effective rate of data flow over a link to a rate that's lower than the link's configured maximum data rate. In particular, mis-configuration of the LTP flow control window can leave transmission capacity unused while LTP engines are awaiting acknowledgments.

- Even if all nodes are correctly configured, a high rate of data loss or corruption due to unexpectedly high R/F interference or underestimated acknowledgment round-trip times may cause an unexpectedly high volume of retransmission traffic. This will displace original bundle transmission, reducing the effective "goodput" data rate on the link.
- Finally, custody transfer may propagate operational problems from one part of the network to other nodes. One result of reduced effective transmission rates is the accumulation of bundles for which nodes have taken custody: the custodial nodes can't destroy those bundles and reclaim the storage space they occupy until custody has been accepted by "downstream" nodes, so abbreviated contacts that prevent the flow of custody acceptances can increase local congestion. This reduces nodes' own ability to take custody of bundles transmitted by "upstream" custodians, increasing queue sizes on those nodes, and so on. In short, custody transfer may itself ultimately impose reactive data triage simply by propagating congestion.

Some level of data triage is essential to cost-effective network utilization, and proactive triage is preferable because its effects can be communicated immediately to users, improving user control over the use of the network. Optimizing an ION-based network therefore amounts to managing for a modicum of proactive data triage and as little reactive data triage as possible. It entails the following:

1. Estimating convergence-layer protocol overhead as accurately as possible, erring (if necessary) on the side of optimism – that is, underestimating a little.

   As an example, suppose the local node uses LTP over CCSDS Telemetry to send bundles. The immediate convergence-layer protocol is LTP, but the total overhead per CL "frame" (in this case, per LTP segment) will include not only the size of the LTP header (nominally 5 bytes) but also the size of the encapsulating space packet header (nominally 6 bytes) and the overhead imposed by the outer encapsulating TM frame.

   Suppose each LTP segment is to be wrapped in a single space packet, which is in turn wrapped in a single TM frame, and Reed-Solomon encoding is applied. An efficient TM frame size is 1115 bytes, with an additional 160 bytes of trailing Reed-Solomon encoding and another 4 bytes of leading pseudo-noise code. The frame would contain a 6-byte TM frame header, a 6-byte space packet header, a 5-byte LTP segment header, and 1098 bytes of some LTP transmission block.

   So the number of "payload bytes per frame" in this case would be 1098 and the number of "overhead bytes per frame" would be 4 + 6 + 6 + 5 + 160 = 181. Nominal total transmission overhead on the link would be 181 / 1279 = about 14%.

2. Synchronizing nodes' clocks as accurately as possible, so that timing margins configured to accommodate clock error can be kept as close to zero as possible.

3. Setting the LTP session limit and block size limit as generously as possible (whenever LTP is at the convergence layer), to assure that LTP flow control does not constrain data flow to rates below those supported by BP rate control.
4. Setting ranges (one-way light times) and queuing delays as accurately as possible, to prevent unnecessary retransmission. Err on the side of pessimism – that is, overestimate a little.
5. Communicating changes in configuration – especially contact plans – to all nodes as far in advance of the time they take effect as possible.
6. Providing all nodes with as much storage capacity as possible for queues of bundles awaiting transmission.

1.9 BP/LTP detail – how it works

Although the operation of BP/LTP in ION is complex in some ways, virtually the entire system can be represented in a single diagram. The interactions among all of the concurrent tasks that make up the node – plus a Remote AMS task or CFDP UT-layer task, acting as the application at the top of the stack – are shown below. (The notation is as used earlier but with semaphores added. Semaphores are shown as small circles, with arrows pointing into them signifying that the semaphores are being given and arrows pointing out of them signifying that the semaphores are being taken.)

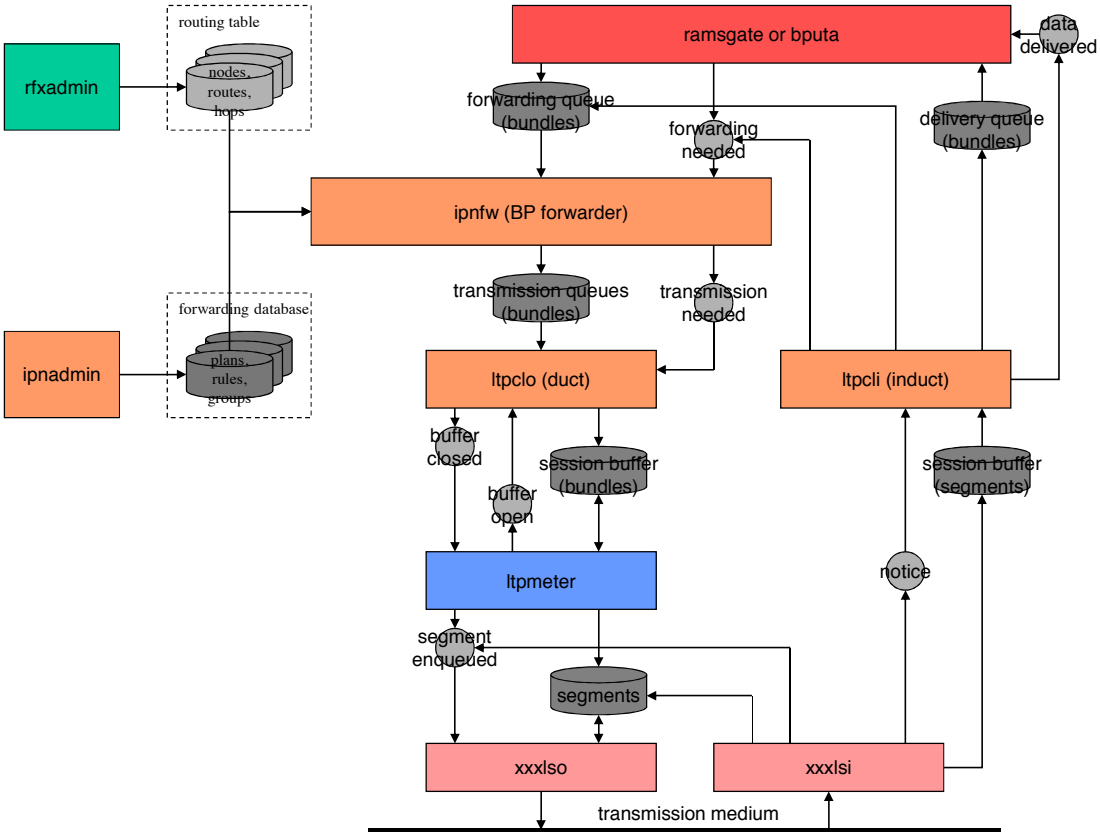


Figure 7 ION node functional overview

Further details of the BP/LTP data structures and flow of control and data appear on the following pages. (For specific details of the operation of the BP and LTP protocols as implemented by the ION tasks, such as the nature of report-initiated retransmission in LTP, please see the protocol specifications. The BP specification is documented in Internet RFC 5050, while the LTP specification is documented in Internet RFC 5326.)

1.9.1 Databases

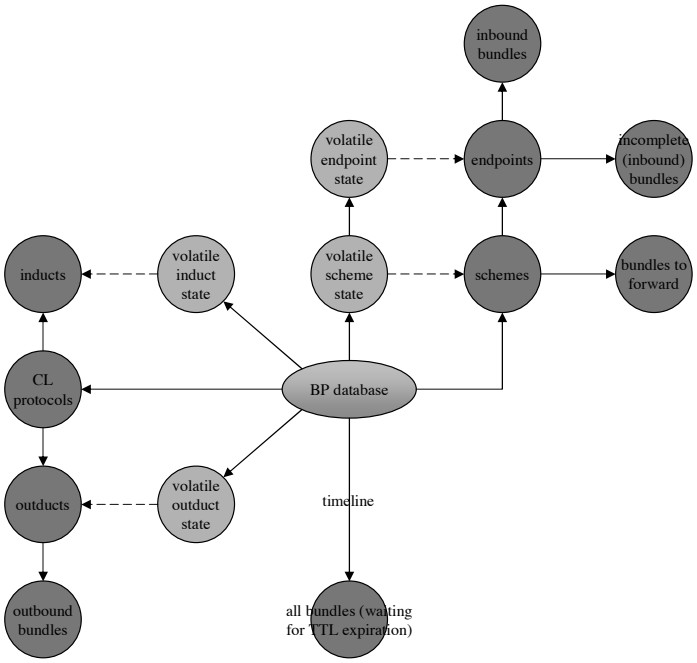


Figure 8 Bundle protocol database

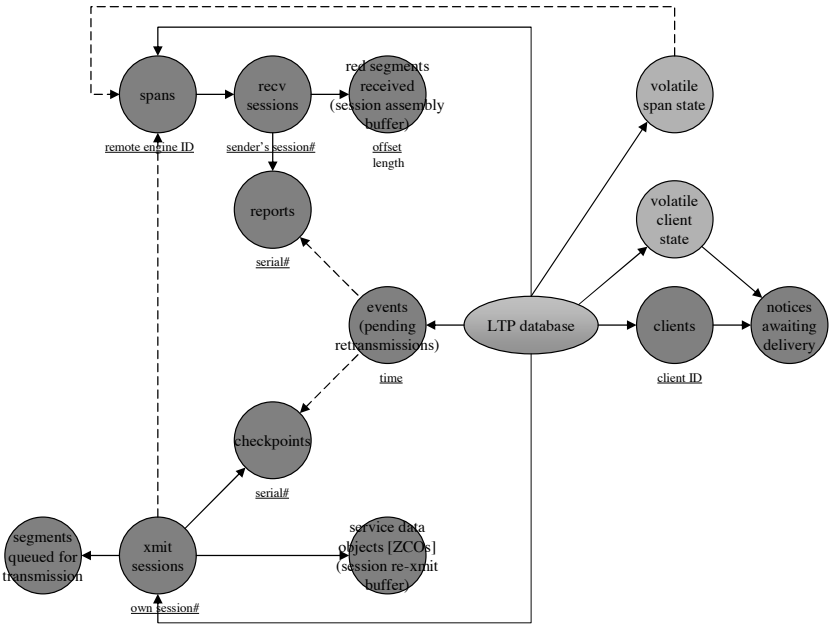
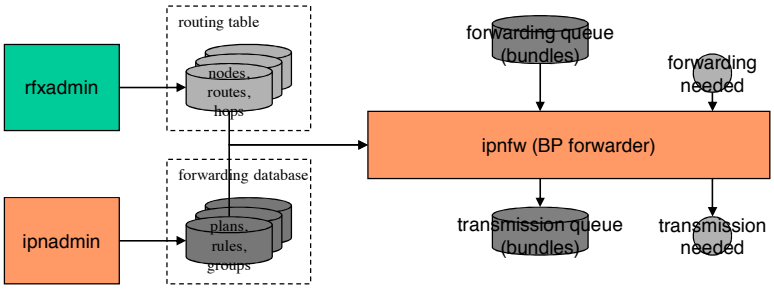


Figure 9 Licklider transmission protocol database

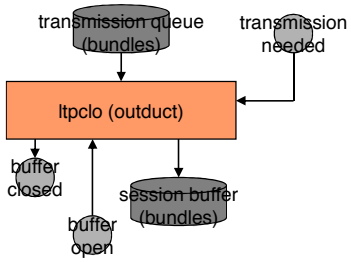
1.9.2 Control and data flow

Bundle Protocol



1. Waits for *forwarding needed* semaphore.
2. Gets bundle from queue.
3. Consults routing table and forwarding table to determine all plausible proximate destinations – routing.
   - A plausible proximate destination is the destination node of the first entry in a contact sequence (a list of concatenated contact periods) ending in a contact period whose destination node is the bundle's destination node and whose start time is less than the bundle's expiration time.
4. Appends bundle to transmission queue (based on priority) for best plausible proximate destination.
5. Gives *transmission needed* semaphore for that transmission queue.

Figure 10 BP forwarder



1. Waits for *buffer open* semaphore (indicating that the link's session buffer has room for the bundle).
2. Waits for *transmission needed* semaphore.
3. Gets bundle from queue, subject to priority.
4. Appends bundle to link's session buffer – aggregation. Buffer size is notionally limited by aggregation size limit, a persistent attribute of the Span object: implicitly, the rate at which we want reports to be transmitted by the destination engine.
5. Gives *buffer closed* semaphore when buffer occupancy reaches the aggregation size limit.

Figure 11 BP convergence layer output

1. Initializes session buffer, gives *buffer open* semaphore.
2. Waits for *buffer closed* semaphore (indicating that the session buffer is ready for transmission).
3. Segments the entire buffer into segments of managed MTU size – fragmentation.
4. Appends all segments to segments queue for immediate transmission.
5. Gives *segment enqueued* semaphore.

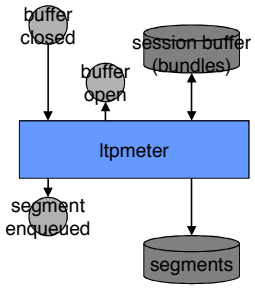


Figure 12 LTP transmission metering

1. Waits for *segment enqueued* semaphore (indicating that there is now something to transmit).
2. Gets segment from queue.
3. Sets retransmission timer if necessary.
4. Transmits the segment using link service protocol.

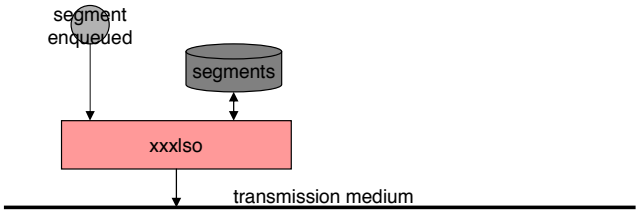


Figure 13 LTP link service output

1. Receives a segment using link service protocol.
2. If data, generates report segment and appends it to queue – reliability. Also inserts data into reception session buffer “red part” and, if that buffer is complete, gives *notice* semaphore to trigger bundle extraction and dispatching by ltpcli.
3. If a report, appends acknowledgement to segments queue.
4. If a report of missing data, recreates lost segments and appends them to queue.
5. Gives *segment enqueued* semaphore.

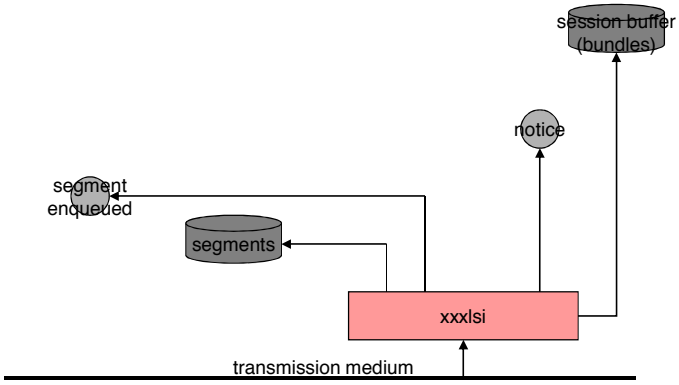


Figure 14 LTP link service input

1.10 Contact Graph Routing (CGR)

CGR is a dynamic routing system that computes routes through a time-varying topology of scheduled communication contacts in a DTN network. It is designed to support operations in a space network based on DTN, but it also could be used in terrestrial applications where operation according to a predefined schedule is preferable to opportunistic communication, as in a low-power sensor network.

The basic strategy of CGR is to take advantage of the fact that, since communication operations are planned in detail, the communication routes between any pair of "bundle agents" in a population of nodes that have all been informed of one another's plans can be inferred from those plans rather than discovered via dialogue (which is impractical over long-one-way-light-time space links).

1.10.1 Contact Plan Messages

CGR relies on accurate contact plan information provided in the form of contact plan messages that currently are only read from **ionrc** files and processed by **ionadmin**, which retains them in a non-volatile contact plan in the RFX database, in ION's SDR data store.

Contact plan messages are of two types: *contact messages* and *range messages*.

Each contact message has the following content:

- The starting UTC time of the interval to which the message pertains.
- The stop time of this interval, again in UTC.
- The Transmitting node number.
- The Receiving node number.
- The planned rate of transmission from node A to node B over this interval, in bytes per second.

Each range message has the following content:

- The starting UTC time of the interval to which the message pertains.
- The stop time of this interval, again in UTC.
- Node number A.
- Node number B.
- The anticipated distance between A and B over this interval, in light seconds.

Note that range messages may be used to declare that the "distance" in light seconds between nodes A and B is **different** in the B→A direction from the distance in the A→B direction. While direct radio communication between A and B will not be subject to such asymmetry, it's possible for connectivity established using other convergence-layer technologies to take different physical paths in different directions, with different signal propagation delays.

1.10.2 Routing Tables

Each node uses Range and Contact messages in the contact plan to build a "routing table" data structure.

The routing table constructed locally by each node in the network is a list of *route lists*, one route list for every other node D in the network that is cited in any Contact or Range in the contact plan.

Each route in the route list for node D identifies a path to destination node D, from the local node, that (a) begins with transmission to one of the local node's neighbors in the network– the initial receiving node for the route, termed the route's *entry node* – and (b) was computed for a specific *payload class*.

A payload class is a payload size limit such that any bundle whose payload's size is less than that limit is known to be forwardable along any route computed for that class. That is, a route computed for payload class value N is guaranteed not to include any contact that has *capacity* – contact duration multiplied by data transmission rate – less than N and therefore can theoretically be guaranteed to accommodate any bundle whose payload's size is no greater than N.

(NOTE that multiple payload classes, enabling more fine-grained route selection in CGR, are not yet supported in ION. For now, routes are computed only for implicit payload class 0.)

For any given route, the contact from the local node to the entry node constitutes the initial transmission segment of the end-to-end path to the destination node. Additionally noted in each route object are all of the other contacts that constitute the remaining segments of the route's end-to-end path.

Each route object also notes the forwarding *cost* for a bundle that is forwarded along this route. In this version of ION, CGR is configured to deliver bundles as early as possible, so best-case final delivery time is used as the cost of a route. Other metrics might be substituted for final delivery time in other CGR implementations. NOTE, however, that if different metrics are used at different nodes along a bundle's end-to-end path it becomes impossible to prevent routing loops that can result in non-delivery of the data.

Finally, each route object also notes the route's termination time, the time after which the route will become moot due to the termination of the earliest-ending contact in the route.

The computed routes for a given destination node are listed in ascending cost order, i.e., the most desirable route apoears first in the list.

1.10.3 Key Concepts

Expiration time

Every bundle transmitted via DTN has a time-to-live (TTL), the length of time after which the bundle is subject to destruction if it has not yet been delivered to its destination. The *expiration time* of a bundle is computed as its creation time plus its TTL. When computing the next-hop destination for a bundle that the local bundle agent

is required to forward, there is no point in selecting a route that can't get the bundle to its final destination prior to the bundle's expiration time.

OWLT margin

One-way light time (OWLT) – that is, distance – is obviously a factor in delivering a bundle to a node prior to a given time. OWLT can actually change during the time a bundle is en route, but route computation becomes intractably complex if we can't assume an OWLT "safety margin" – a maximum delta by which OWLT between any pair of nodes can change during the time a bundle is in transit between them.

We assume that the maximum rate of change in distance between any two nodes in the network is about 150,000 miles per hour, which is about 40 miles per second. (This was the speed of the Helios spacecraft, the fastest man-made object launched to date.)

At this speed, the distance between any two nodes that are initially separated by a distance of N light seconds will increase by a maximum of 40 miles per second of transit. This will result in data arrival no later than roughly (N + Q) seconds after transmission – where the "OWLT margin" value Q is (40 * N) divided by 186,000 – rather than just N seconds after transmission as would be the case if the two nodes were stationary relative to each other. When computing the expected time of arrival of a transmitted bundle we simply use N + Q, the most pessimistic case, as the anticipated total in-transit time.

Capacity

The *capacity* of a contact is the product of its data transmission rate (in bytes per second) and its duration (stop time minus start time, in seconds).

Estimated capacity consumption

The size of a bundle is the sum of its payload size and its header size[5], but bundle size is not the only lien on the capacity of a contact. The total estimated capacity consumption (or "ECC") for a bundle that is queued for transmission via some outduct is a more lengthy computation.

For each recognized convergence-layer protocol, we can estimate the number of bytes of "overhead" (that is, data that serves the purposes of the protocol itself rather than the user application that is using it) for each frame of convergence-layer protocol transmission. If the convergence layer protocol were UDP/IP over the Internet, for example, we might estimate the convergence layer overhead per frame to be 100 bytes – allowing for the nominal sizes of the UDP, IP, and Ethernet or SONET overhead for each IP packet.

We can estimate the number of bundle bytes per CL protocol frame as the total size of each frame less the per-frame convergence layer overhead. Continuing the example begun above, we might estimate the number of bundle bytes per frame to be 1400, which is the standard MTU size on the Internet (1500 bytes) less the estimated convergence layer overhead per frame

[5] The minimum size of an ION bundle header is 26 bytes. Adding extension blocks (such as those that effect the Bundle Security Protocol) will increase this figure.

We can then estimate the total number of frames required for transmission of a bundle of a given size: this number is the bundle size divided by the estimated number of bundle bytes per CL protocol frame, rounded up.

The estimated total convergence layer overhead for a given bundle is, then, the per-frame convergence layer overhead multiplied by the total number of frames required for transmission of a bundle of that size

Finally the ECC for that bundle can be computed as the sum of the bundle's size and its estimated total convergence layer overhead.

Residual capacity

The *residual capacity* of a given contact between the local node and one of its neighbors, as computed for a given bundle, is the sum of the capacities of that contact and all prior scheduled contacts between the local node and that neighbor, less the sum of the ECCs of all bundles with priority equal to or higher than the priority of the subject bundle that are currently queued on the outduct for transmission to that neighbor.

Plausible opportunity

A *plausible opportunity* for transmitting a given bundle to some neighboring node is defined as a contact whose residual capacity is at least equal to the bundle's ECC. That is, if the capacity of a given contact is already fully subscribed, when computing routes for the next bundle there is no purpose served by assuming transmission during that contact.

Excluded neighbors

A neighboring node C that refuses custody of a bundle destined for some remote node D is termed an *excluded neighbor* for (that is, with respect to computing routes to) D. So long as C remains an excluded neighbor for D, no bundles destined for D will be forwarded to C – except that occasionally (once per lapse of the RTT between the local node and C) a custodial bundle destined for D will be forwarded to C as a "probe bundle". C ceases to be an excluded neighbor for D as soon as it accepts custody of a bundle destined for D.

Critical bundles

A Critical bundle is one that absolutely has got to reach its destination and, moreover, has got to reach that destination as soon as is physically possible[6].

For an ordinary non-Critical bundle, the CGR dynamic route computation algorithm uses the routing table to select a single neighboring node to forward the bundle through. It is possible, though, that due to some unforeseen delay the selected neighbor may prove to be a sub-optimal forwarder: the bundle might arrive later than it would have if another neighbor had been selected, or it might not even arrive at all.

[6] In ION, all bundles are by default non-critical. The application can indicate that data should be sent in a Critical bundle by setting the BP_MINIMUM_LATENCY flag in the "extended class of service" parameter, but this feature is an ION extension that is not supported by other BP implementations at the time of this writing.

For Critical bundles, the CGR dynamic route computation algorithm causes the bundle to be inserted into the outbound transmission queues for transmission to all neighboring nodes that can plausibly forward the bundle to its final destination. The bundle is therefore guaranteed to travel over the most successful route, as well as over all other plausible routes. Note that this may result in multiple copies of a Critical bundle arriving at the final destination.

1.10.4 Dynamic Route Selection Algorithm

Given a bundle whose destination is node D, we proceed as follows.

First, if no contacts in the contact plan identify transmission to node D, then we cannot use CGR to find a route for this bundle; CGR route selection is abandoned.

Next, if the contact plan has been modified in any way since routes were computed for any nodes, we discard all routes for all nodes and authorize route recomputaton. (The contact plan changes may have invalidated any or all of those earlier computations.)

We create an empty list of Proximate Nodes (network neighbors) to send the bundle to.

We create a list of Excluded Nodes, i.e., nodes through which we will not compute a route for this bundle. The list of Excluded Nodes is initially populated with:

- the node from which the bundle was directly received (so that we avoid cycling the bundle between that node and the local node) – unless the Dynamic Route Selection Algorithm is being re-applied due to custody refusal as discussed later;
- all excluded neighbors for the bundle's final destination node.

If all routes computed for node D have been discarded due to contact plan modification, then we must compute a new list of all routes from the local node to D. To do so:

- We construct an abstract contact graph, a directed acyclic graph whose root is a notional contact from the local node to itself and whose other vertices are all other contacts representing transmission "from" some node such that a contact "to" that node already exists in the graph, excluding contacts representing transmission "to" some node such that a contact "from" that node already exists in the graph. A terminal vertex is also included in the graph, constituting a notional contact from node D to itself.
- We perform several series of Dijkstra searches within this graph, one series of searches for each payload class. On each search we find the lowest-cost route that begins at the root of the graph and ends at the terminal vertex. Each time a route is computed, we add it to the node's list of routes and then remove the route's initial contact from the contact graph before searching for the next best route. Each search series is terminated as soon as a search fails to find a route.
    - During any search, every contact whose capacity is less than the applicable payload class for the search is ignored.
    - The lowest-cost route computed during a search is the one that is found to have the earliest best-case delivery time, where the best-case delivery time characterizing a route is given by the time at which a bundle would arrive

at node D if transmitted at the earliest possible moment of the last contact in the route prior to the terminal vertex.

    - Any contact whose end time is before the earliest possible time that the bundle could arrive at the contact's sending node is ignored.
    - The earliest possible arrival time for the bundle on a given contact is pessimistically computed as the sum of the bundle's earliest possible transmission time plus the range in light seconds from the contact's sending node to its receiving node, plus the applicable one-way light time margin.
    - The earliest possible transmission time for the bundle on a given contact is the start time of the contact or bundle's earliest possible arrival time at the contact's sending node, whichever is later.

- If node D's list of routes is still empty, then we cannot use CGR to find a route for this bundle; CGR route selection is abandoned.

We next examine all of the routes that are currently computed for transmission of bundles to node D.

- Any route whose termination time is in the past is deleted from the list, and all contacts in that route whose termination time is in the past are also deleted. But if the end time of that route's initial contact is still in the future, we run another Dijkstra search to compute the best route (for the deleted route's payload class) given the remaining contacts, excluding all contacts that are initial contacts of other routes that have not yet been deleted; if this search finds a route, the new route is inserted into the appropriate location in the list.
- Any route whose best-case final delivery time is after the bundle's expiration time is ignored, as is any route whose entry node is in the list of Excluded Nodes. Any route that includes a contact whose capacity is less than the bundle's payload size is also ignored. Loopback routes are also ignored unless the local node is the bundle's final destination.
- For each route, the aggregate radiation time for this bundle on this route is computed by summing the product of payload size and contact transmission rate over all contacts in the route. Any route for which the sum of best-case delivery time and aggregate radiation time is after the bundle's expiration time is ignored.

For each route that is not ignored:

- We locate in the unicast forwarding database the egress plan for the route's entry node. From this directive we infer the convergence-layer "outduct" on which the bundle would be sent if transmitted to that node. We then determine whether or not the bundle could be transmitted during the initial contact of this route. There are three criteria:
    - If the outduct is currently "blocked" due to a detected or asserted loss of connectivity, then the route cannot be selected.

    - If the bundle cannot be fragmented and its payload's size exceeds the outduct's payload size limit, then the route cannot be selected.
    - If the contact is not a "plausible opportunity" (as defined earlier) for transmission of this bundle, then the route cannot be selected.
- If the route is eligible for selection:
    - If the route's entry node has not yet been added to the list of Proximate Nodes for this bundle, then it is added to that list. Associated with the entry node number in this list entry are the best-case final delivery time of the route, the total number of "hops" in the route's end-to-end path, and the *forfeit time* for transmission to this node. Forfeit time is the route's termination time, the time by which the bundle must have been transmitted to this node in order to have any chance of being forwarded on this route.
    - Otherwise (i.e., this route's entry node is already in the Proximate Nodes list), if the route's best-case final delivery time is earlier than that of the existing Proximate Nodes list entry for this node, then the earlier time replaces that later time; if the delivery times are equal but the route's hop count is less than that of the existing entry, then the smaller hop count replaces the larger one; if either of these changes are made, then this route's forfeit time replaces the list entry's current forfeit time.

If, at the end of this procedure, the Proximate Nodes list is empty, then we have been unable to use CGR to find a route for this bundle; CGR route selection is abandoned.

Otherwise:

- If the bundle is flagged as a critical bundle, then a cloned copy of this bundle is enqueued for transmission on the outduct to every node in the Proximate Nodes list.
- Otherwise, the bundle is enqueued for transmission on the outduct to the most preferred neighbor in the Proximate Nodes list:
    - If one of the nodes in this list is associated with a best-case delivery time that is earlier than that of all other nodes in the list, then it is the most preferred neighbor.
    - Otherwise, if one of the nodes with the earliest best-case delivery time is associated with a smaller hop count than every other node with the same best-case delivery time, then it is the most preferred neighbor.
    - Otherwise, the node with the smallest node number among all nodes with the earliest best-case delivery time and smallest hop count is arbitrarily chosen as the most preferred neighbor.

1.10.5 Exception Handling

Conveyance of a bundle from source to destination through a DTN can fail in a number of ways, many of which are best addressed by means of the Delivery Assurance mechanisms described earlier. Failures in Contact Graph Routing, specifically, occur

when the expectations on which routing decisions are based prove to be false. These failures of information fall into two general categories: contact failure and custody refusal.

1) Contact failure

   A scheduled contact between some node and its neighbor on the end-to-end route may be initiated later than the originally scheduled start time, or be terminated earlier than the originally scheduled stop time, or be canceled altogether. Alternatively, the available capacity for a contact might be overestimated due to, for example, diminished link quality resulting in unexpectedly heavy retransmission at the convergence layer. In each of these cases, the anticipated transmission of a given bundle during the affected contact may not occur as planned: the bundle might expire before the contact's start time, or the contact's stop time might be reached before the bundle has been transmitted.

   For a non-Critical bundle, we handle this sort of failure by means of a timeout: if the bundle is not transmitted prior to the forfeit time for the selected Proximate Node, then the bundle is removed from its outbound transmission queue and the Dynamic Route Computation Algorithm is re-applied to the bundle so that an alternate route can be computed.

2) Custody refusal

   A node that receives a bundle may find it impossible to forward it, for any of several reasons: it may not have enough storage capacity to hold the bundle, it may be unable to compute a forward route (static, dynamic, or default) for the bundle, etc. Such bundles are simply discarded, but discarding any such bundle that is marked for custody transfer will cause a custody refusal signal to be returned to the bundle's current custodian.

   When the affected bundle is non-Critical, the node that receives the custody refusal re-applies the Dynamic Route Computation Algorithm to the bundle so that an alternate route can be computed – except that in this event the node from which the bundle was originally directly received is omitted from the initial list of Excluded Nodes. This enables a bundle that has reached a dead end in the routing tree to be sent back to a point at which an altogether different branch may be selected.

For a Critical bundle no mitigation of either sort of failure is required or indeed possible: the bundle has already been queued for transmission on all plausible routes, so no mechanism that entails re-application of CGR's Dynamic Route Computation Algorithm could improve its prospects for successful delivery to the final destination. However, in some environments it may be advisable to re-apply the Dynamic Route Computation Algorithm to all Critical bundles that are still in local custody whenever a new Contact is added to the contact graph: the new contact may open an additional forwarding opportunity for one or more of those bundles.

1.10.6 Remarks

The CGR routing procedures respond dynamically to the changes in network topology that the nodes are able know about, i.e., those changes that are subject to mission operations control and are known in advance rather than discovered in real time. This dynamic responsiveness in route computation should be significantly more effective and less expensive than static routing, increasing total data return while at the same time reducing mission operations cost and risk.

Note that the non-Critical forwarding load across multiple parallel paths should be balanced automatically:

- Initially all traffic will be forwarded to the node(s) on what is computed to be the best path from source to destination.
- At some point, however, a node on that preferred path may have so much outbound traffic queued up that no contacts scheduled within bundles' lifetimes have any residual capacity. This can cause forwarding to fail, resulting in custody refusal.
- Custody refusal causes the refusing node to be temporarily added to the current custodian's excluded neighbors list for the affected final destination node. If the refusing node is the only one on the path to the destination, then the custodian may end up sending the bundle back to its upstream neighbor. Moreover, that custodian node too may begin refusing custody of bundles subsequently sent to it, since it can no longer compute a forwarding path.
- The upstream propagation of custody refusals directs bundles over alternate paths that would otherwise be considered suboptimal, balancing the queuing load across the parallel paths.
- Eventually, transmission and/or bundle expiration at the oversubscribed node relieves queue pressure at that node and enables acceptance of custody of a "probe" bundle from the upstream node. This eventually returns the routing fabric to its original configuration.

Although the route computation procedures are relatively complex they are not computationally difficult. The impact on computation resources at the vehicles should be modest.

1.11 LTP Timeout Intervals

Suppose we've got Earth ground station ES that is currently in view of Mars but will be rotating out of view ("Mars-set") at some time T1 and rotating back into view ("Mars-rise") at time T3. Suppose we've also got Mars orbiter MS that is currently out of the shadow of Mars but will move behind Mars at time T2, emerging at time T4. Let's also suppose that ES and MS are 4 light-minutes apart (Mars is at its closest approach to Earth). Finally, for simplicity, let's suppose that both ES and MS want to be communicating at every possible moment (maximum link utilization) but never want to waste any electricity.

Neither ES nor MS wants to be wasting power on either transmitting or receiving at a time when either Earth or Mars will block the signal.

ES will therefore stop transmitting at either T1 or (T2 - 4 minutes), whichever is earlier; call this time T_{et0}. It will stop receiving – that is, power off the receiver – at either T1 or (T2 + 4 minutes), whichever is earlier; call this time T_{er0}. It will resume transmitting at either T3 or (T4 - 4 minutes), whichever is later, and it will resume reception at either T3 or (T4 + 4 minutes), whichever is later; call these times T_{et1} and T_{er1}.

Similarly, MS will stop transmitting at either T2 or (T1 - 4 minutes), whichever is earlier; call this time T_{mt0}. It will stop receiving – that is, power off the receiver – at either T2 or (T1 + 4 minutes), whichever is earlier; call this time T_{mr0}. It will resume transmitting at either T4 or (T3 - 4 minutes), whichever is later, and it will resume reception at either T4 or (T3 + 4 minutes), whichever is later; call these times T_{mt1} and T_{mr1}.

By making sure that we don't transmit when the signal would be blocked, we guarantee that anything that is transmitted will arrive at a time when it can be received. Any reception failure is due to data corruption en route.

So the moment of transmission of an acknowledgment to any message is always equal to the moment the original message was sent plus some imputed outbound queuing delay QO1 at the sending node, plus 4 minutes, plus some imputed inbound and outbound queuing delay QI1 + QO2 at the receiving node. The nominally expected moment of reception of this acknowledgment is that moment of transmission plus 4 minutes, plus some imputed inbound queuing delay QI2 at the original sending node. That is, the timeout interval is 8 minutes + QO1 + QI1 + QO2 + QO2 – *unless* this moment of acknowledgment transmission is during an interval when the receiving node is not transmitting, for whatever reason. In this latter case, we want to suspend the acknowledgment timer during any interval in which we know the remote node will not be transmitting. More precisely, we want to add to the timeout interval the time difference between the moment of message arrival and the earliest moment at which the acknowledgment could be sent, i.e., the moment at which transmission is resumed[7].

[7] If we wanted to be extremely accurate we could also subtract from the timeout interval the imputed inbound queuing delay QI, since inbound queuing would presumably be completed during the interval in which transmission was suspended. But since we're guessing at the queuing delays anyway, this adjustment doesn't make a lot of sense.

So the timeout interval Z computed at ES for a message sent to MS at time T_X is given by:

$$Z = QO1 + 8 + QI1 + ((T_A = T_X + 4) > T_{mt0} \;\&\&\; T_A < T_{mt1}) \;?\; T_{mt1} - T_A : 0) + QI2 + QO2;$$

This can actually be computed in advance (at time T_X) if T1, T2, T3, and T4 are known and are exposed to the protocol engine.

If they are not exposed, then Z must initially be estimated to be (2 * the one-way light time) + QI + QO. The timer for Z must be dynamically suspended at time T_{mt0} in response to a state change as noted by **ltpclock**. Finally, the timer must be resumed at time T_{mt1} (in response to another state change as noted by **ltpclock**), at which moment the correct value for Z can be computed.

1.12 CFDP

The ION implementation of CFDP is very simple, because only Class-1 (Unacknowledged) functionality is implemented: the store-and-forward routing performed by Bundle Protocol makes the CFDP Extended Procedures unnecessary and the inter-node reliability provided by the CL protocol underneath BP – in particular, by LTP – makes the CFDP Acknowledged Procedures unnecessary. All that CFDP is required to do is segment and reassemble files, interact with the underlying Unitdata Transfer layer – BP/LTP – to effect the transmission and reception of file data segments, and handle CFDP metadata including filestore requests. CFDP-ION does all this, including support for cancellation of a file transfer transaction by cancellation of the transmission of the bundles encapsulating the transaction's protocol data units.

Note that all CFDP data transmission is "by reference", via the ZCO system, rather than "by value": the retransmission buffer for a bundle containing CFDP file data is an extent of the original file itself, not a copy retained in the ION database, and data received in bundles containing CFDP PDU is written immediately to the appropriate location in the reconstituted file rather than stored in the ION database. This minimizes the space needed for the database. In general, file transmission via CFDP is the most memory-efficient way to use ION in flight operations.

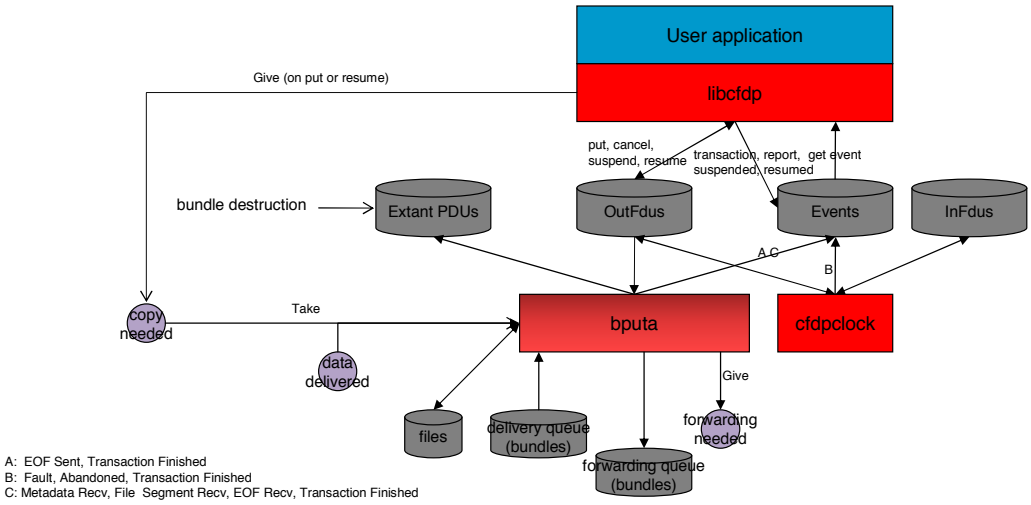


Figure 15 A CFDP-ION entity

1.13 Additional Figures for Manual Pages

1.13.1 list data structures (lyst, sdrlist, smlist)

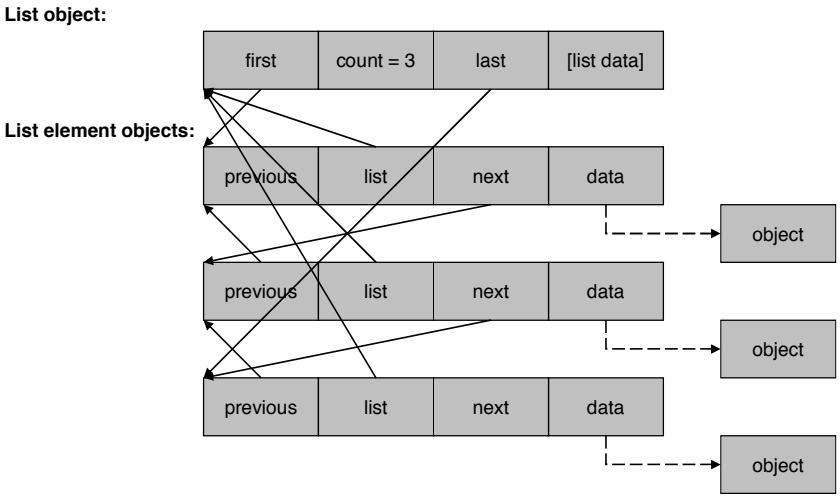


Figure 16 ION list data structures

1.13.2 psm partition structure

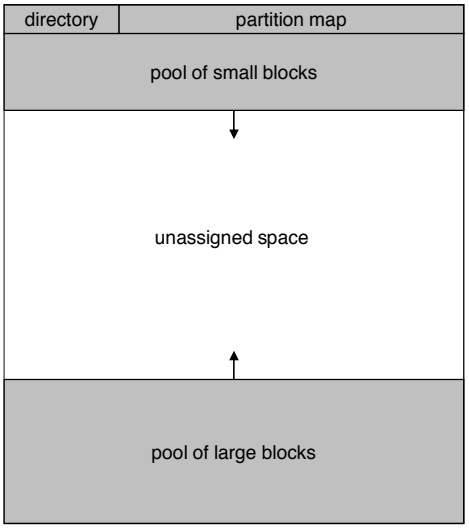


Figure 17 psm partition structure

1.13.3 psm and sdr block structures

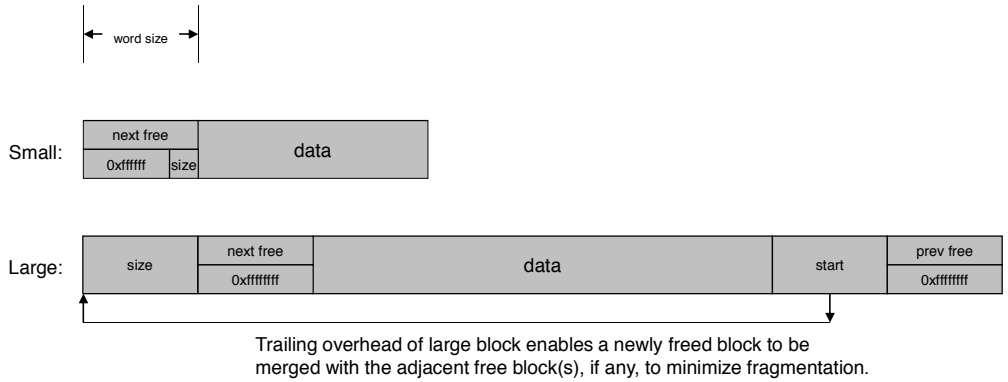


Figure 18 psm and sdr block structures

1.13.4 sdr heap structure

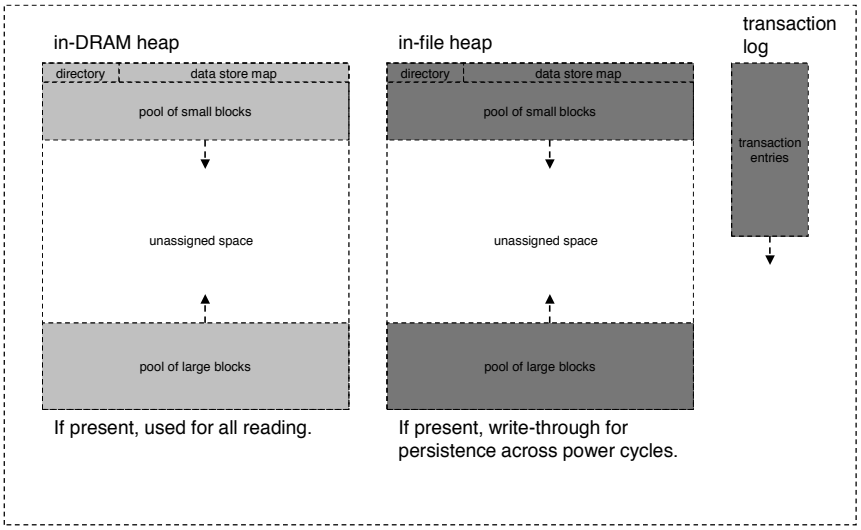


Figure 19 sdr heap structure

2 Operation

One compile-time option is applicable to all ION packages: the platform selection parameters –DVXWORKS and –DRTEMS affect the manner in which most task instantiation functions are compiled. For VXWORKS and RTEMS, these functions are compiled as library functions that must be identified by name in the platform's symbol table, while for Unix-like platforms they are compiled as `main()` functions.

2.1 Interplanetary Communication Infrastructure (ICI)

2.1.1 Compile-time options

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, VxWorks 5.4, or RTEMS. (For these latter 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 10 for Cygwin and 128 otherwise. (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 6 for Cygwin and 250 otherwise. (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 60 for Cygwin and 32000 otherwise. (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).

2.1.2 Build

To build ICI for a given deployment platform:

1. Decide where you want ION's executables, libraries, header files, etc. to be installed. The ION makefiles all install their build products to subdirectories (named **bin**, **lib**, **include**, **man**, **man/man1**, **man/man3**, **man/man5**) of an ION root directory, which by default is the directory named **/opt**. If you wish to use the default build configuration, be sure that the default directories (**/opt/bin**, etc.) exist; if not, select another ION root directory name – this document will refer to it as **$OPT** – and create the subdirectories as needed. In any case, make sure that you have read, write, and execute permission for all of the ION installation directories and that:
    - The directory **/$OPT/bin** is in your execution path.
    - The directory **/$OPT/lib** is in your $LD_LOADLIB_PATH.
2. Edit the Makefile in **ion/ici**:
    - Make sure PLATFORMS is set to the appropriate platform name, e.g., x86-redhat, sparc-sol9, etc.
    - Set OPT to your ION root directory name, if other than "/opt".
3. Then:

```
cd ion/ici
make
make install
```

2.1.3 Configure

Three types of files are used to provide the information needed to perform global configuration of the ION protocol stack: the ION system configuration (or **ionconfig**) file, the ION administration command (**ionrc**) file, and the ION security configuration (**ionsecrc**) file. For details, see the man pages for ionconfig(5), ionrc(5), and ionsecrc(5) in Appendix A.

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 nodes 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().

2.1.4 Run

The executable programs used in operation of the ici component of ION include:

- The **ionadmin** system configuration utility and **ionsecadmin** security configuration utility, invoked at node startup time and as needed thereafter.
- The **rfxclock** background daemon, which effects scheduled network configuration events.
- The **sdrmend** system repair utility, invoked as needed.
- The **sdrwatch** and **psmwatch** utilities for resource utilization monitoring, invoked as needed.

Each time it is executed, **ionadmin** computes a new congestion forecast and, if a congestion collapse is predicted, invokes the node's congestion alarm script (if any). **ionadmin** also establishes the node number for the local node and starts/stops the **rfxclock** task, among other functions. For further details, see the man pages for ionadmin(1), ionsecadmin(1), rfxclock(1), sdrmend(1), sdrwatch(1), and psmwatch(1) in Appendix A.

2.1.5 Test

Six test executables are provided to support testing and debugging of the ICI component of ION:

- The **file2sdr** and **sdr2file** programs exercise the SDR system.
- The **psmshell** program exercises the PSM system.
- The **file2sm**, **sm2file**, and **smlistsh** programs exercise the shared-memory linked list system.

For details, see the man pages for file2sdr(1), sdr2file(1), psmshell(1), file2sm(1), sm2file(1), and smlistsh(1) in Appendix A.

2.2 Licklider Transmission Protocol (LTP)

2.2.1 Build

To build LTP:

1. Make sure that the "ici" component of ION has been built for the platform on which you plan to run LTP.
2. Edit the Makefile in **ion/ltp**:
   - As for ici, make sure PLATFORMS is set to the name of platform on which you plan to run LTP.
   - Set OPT to the directory containing the bin, lib, include, etc. directories used for building ici.
3. Then:

```
cd ion/ltp
make
make install
```

2.2.2 Configure

The LTP administration command (**ltprc**) file provides the information needed to configure LTP on a given ION node. For details, see the man page for ltprc(5) in Appendix A.

2.2.3 Run

The executable programs used in operation of the ltp component of ION include:

- The **ltpadmin** protocol configuration utility, invoked at node startup time and as needed thereafter.
- The **ltpclock** background daemon, which effects scheduled LTP events such as segment retransmissions.
- The **ltpmeter** block management daemon, which segments blocks and effects LTP flow control.
- The **udplsi** and **udplso** link service input and output tasks, which handle transmission of LTP segments encapsulated in UDP datagrams (mainly for testing purposes).

ltpadmin starts/stops the **ltpclock** task and, as mandated by configuration, the **udplsi** and **udplso** tasks.

For details, see the man pages for ltpadmin(1), ltpclock(1), ltpmeter(1), udplsi(1), and udplso(1) in Appendix A.

2.2.4 Test

Two test executables are provided to support testing and debugging of the LTP component of ION:

- **ltpdriver** is a continuous source of LTP segments.
- **ltpcounter** is an LTP block receiver that counts blocks as they arrive.

For details, see the man pages for ltpdriver(1) and ltpcounter(1) in Appendix A.

2.3 Bundle Protocol (BP)

2.3.1 Compile-time options

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.

ENABLE_BPACS

This option causes Aggregate Custody Signaling source code to be included in the build. ACS is alternative custody transfer signaling mechanism that sharply reduces the volume of custody acknowledgment traffic.

ENABLE_IMC

This option causes IPN Multicast source code to be included in the build. IMC is discussed in section 1.8.4 above.

2.3.2 Build

To build BP:

1. Make sure that the "ici" and "dg" (see below) components of ION have been built for the platform on which you plan to run BP.
2. Edit the Makefile in **ion/bp**:
   - As for ici, make sure PLATFORMS is set to the name of platform on which you plan to run BP.
   - Set OPT to the directory containing the bin, lib, include, etc. directories used for building ici.
3. Then:

```
cd ion/bp
make
make install
```

2.3.3 Configure

The BP administration command (**bprc**) file provides the information needed to configure generic BP on a given ION node. The IPN scheme administration command (**ipnrc**) file provides information that configures static and default routes for endpoints whose IDs conform to the "ipn" scheme. The DTN scheme administration command (**dtn2rc**) file provides information that configures static and default routes for endpoints whose IDs conform to the "dtn" scheme, as supported by the DTN2 reference implementation. For details, see the man pages for bprc(5), ipnrc(5), and dtn2rc(5) in Appendix A.

2.3.4 Run

The executable programs used in operation of the bp component of ION include:

- The **bpadmin, ipnadmin,** and **dtn2admin** protocol configuration utilities, invoked at node startup time and as needed thereafter.
- The **bpclock** background daemon, which effects scheduled BP events such as TTL expirations and which also implements rate control.
- The **ipnfw** and **dtn2fw** forwarding daemons, which compute routes for bundles addressed to "ipn"-scheme and "dtn"-scheme endpoints, respectively.
- The **ipnadminep** and **dtn2adminep** administrative endpoint daemons, which handle custody acceptances, custody refusals, and status messages.
- The **brsscla** (server) and **brsccla** (client) Bundle Relay Service convergence-layer adapters.
- The **tcpcli** (input) and **tcpclo** (output) TCP convergence-layer adapters.

- The **udpcli** (input) and **udpclo** (output) UDP convergence-layer adapters.
- The **ltpcli** (input) and **ltpclo** (output) LTP convergence-layer adapters.
- The **dgrcla** Datagram Retransmission convergence-layer adapter.
- The **bpsendfile** utility, which sends a file of arbitrary size, encapsulated in a single bundle, to a specified BP endpoint.
- The **bpstats** utility, which prints a snapshot of currently accumulated BP processing statistics on the local node.
- The **bptrace** utility, which sends a bundle through the network to enable a forwarding trace based on bundle status reports.
- The **lgsend** and **lgagent** utilities, which are used for remote administration of ION nodes.
- The **hmackeys** utility, which can be used to create hash keys suitable for use in bundle authentication blocks and BRS convergence-layer protocol connections.

bpadmin starts/stops the **bpclock** task and, as mandated by configuration, the **ipnfw**, **dtn2fw**, **ipnadminep**, **dtn2adminep**, **brsscla**, **brsccla**, , **tcpcli**, **tcpclo**, **udpcli**, **udpclo**, **ltpcli**, **ltpclo**, and **dgrcla** tasks.

For details, see the man pages for bpadmin(1),ipnadmin(1), dtn2admin(1), bpclock(1), ipnfw(1), dtn2fw(1), ipnadminep(1), dtn2adminep(1), brsscla(1), brsccla(1), tcpcli(1), tcpclo(1), udpcli(1), udpclo(1), ltpcli(1), ltpclo(1), dgrcla(1), bpsendfile(1), bpstats(1), bptrace(1), lgsend(1), lgagent(1), and hmackeys(1) in Appendix A.

2.3.5 Test

Five test executables are provided to support testing and debugging of the BP component of ION:

- **bpdriver** is a continuous source of bundles.
- **bpcounter** is a bundle receiver that counts bundles as they arrive.
- **bpecho** is a bundle receiver that sends an "echo" acknowledgment bundle back to bpdriver upon reception of each bundle.
- **bpsource** is a simple console-like application for interactively sending text strings in bundles to a specified DTN endpoint, nominally a **bpsink** task.
- **bpsink** is a simple console-like application for receiving bundles and printing their contents.

For details, see the man pages for bpdriver(1), bpcounter(1), bpecho(1), bpsource(1), and bpsink(1) in Appendix A.

2.4 Datagram Retransmission (DGR)

2.4.1 Build

To build DGR:

1. Make sure that the "ici" component of ION has been built for the platform on which you plan to run DGR.
2. Edit the Makefile in **ion/dgr**:
   - As for ici, make sure PLATFORMS is set to the name of platform on which you plan to run DGR.
   - Set OPT to the directory containing the bin, lib, include, etc. directories used for building ici.
3. Then:

```
cd ion/dgr
make
make install
```

2.4.2 Configure

No additional configuration files are required for the operation of the DGR component of ION.

2.4.3 Run

No runtime executables are required for the operation of the DGR component of ION.

2.4.4 Test

Two test executables are provided to support testing and debugging of the DGR component of ION:

- **file2dgr** repeatedly reads a file of text lines and sends copies of those text lines via DGR to **dgr2file**, which writes them to a copy of the original file.

For details, see the man pages for file2dgr(1) and dgr2file(1) in Appendix A.

2.5 Asynchronous Message Service (AMS)

2.5.1 Compile-time options

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.

2.5.2 Build

To build AMS:

1. Make sure that the "bp" component of ION has been built for the platform on which you plan to run AMS.
2. Edit the Makefile in **ion/cfdp**:
   - Just as for bp, make sure PLATFORMS is set to the name of platform on which you plan to run AMS.
   - Set OPT to the directory containing the bin, lib, include, etc. directories used for building bp.
3. Then:

```
cd ion/ams
make
make install
```

2.5.3 Configure

There is no central configuration of AMS; each AMS entity (configuration server, registrar, or application module) is individually configured at the time its initial MIB is

loaded at startup. For details of MIB file syntax, see the man pages for amsrc(5) and amsxml(5) in Appendix A.

2.5.4 Run

The executable programs used in operation of the AMS component of ION include:

- The **amsd** background daemon, which serves as configuration server and/or as the registrar for a single application cell.
- The **ramsgate** application module, which serves as the Remote AMS gateway for a single message space.
- The **amsstop** utility, which terminates all AMS operation throughout a single message space.
- The **amsmib** utility, which announces supplementary MIB information to selected subsets of AMS entities without interrupting the operation of the message space.

For details, see the man pages for amsd(1), ramsgate(1), amsstop(1), and amsmib(1) in Appendix A.

2.5.5 Test

Seven test executables are provided to support testing and debugging of the AMS component of ION:

- **amsbenchs** is a continuous source of messages.
- **amsbenchr** is a message receiver that calculates bundle transmission performance statistics.
- **amshello** is an extremely simple AMS "hello, world" demo program – a self-contained distributed application in a single source file of about seventy lines.
- **amsshell** is a simple console-like application for interactively publishing, sending, and announcing text strings in messages.
- **amslog** is a simple console-like application for receiving messages and piping their contents to stdout.
- **amslogprt** is a pipeline program that simply prints AMS message contents piped to it from amslog.
- **amspubsub** is a pair of functions for rudimentary testing of AMS functionality in a VxWorks environment.

For details, see the man pages for amsbenchs(1), amsbenchr(1), amshello(1), amsshell(1), amslog(1), amslogprt(1), amspub(1), and amssub(1) in Appendix A.

For further operational details of the AMS system, please see sections 4 and 5 of the AMS Programmer's Guide.

2.6 CCSDS File Delivery Protocol (CFDP)

2.6.1 Compile-time options

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.

2.6.2 Build

To build CFDP:

1. Make sure that the "bp" component of ION has been built for the platform on which you plan to run CFDP.
2. Edit the Makefile in **ion/cfdp**:
   - Just as for bp, make sure PLATFORMS is set to the name of platform on which you plan to run CFDP.
   - Set OPT to the directory containing the bin, lib, include, etc. directories used for building bp.
3. Then:

```
cd ion/cfdp
make
make install
```

2.6.3 Configure

The CFDP administration command (**cfdprc**) file provides the information needed to configure CFDP on a given ION node. For details, see the man page for cfdprc(5) in Appendix A.

2.6.4 Run

The executable programs used in operation of the CFDP component of ION include:

- The **cfdpadmin** protocol configuration utility, invoked at node startup time and as needed thereafter.
- The **cfdpclock** background daemon, which effects scheduled CFDP events such as check timer expirations. The **cfdpclock** task also effects CFDP transaction cancellations, by canceling the bundles encapsulating the transaction's protocol data units.
- The **bputa** UT-layer input/output task, which handles transmission of CFDP PDUs encapsulated in bundles.

cfdpadmin starts/stops the **cfdpclock** task and, as mandated by configuration, the **bputa** task.

For details, see the man pages for cfdpadmin(1), cfdpclock(1), and bputa(1) in Appendix A.

2.6.5 Test

A single executable, **cfdptest**, is provided to support testing and debugging of the DGR component of ION. For details, see the man page for cfdptest(1) in Appendix A.

2.7 Bundle Streaming Service (BSS)

2.7.1 Compile-time options

Defining the following macro, by setting a parameter that is provided to the C compiler (e.g., –DWINDOW=10000), will alter the functionality of BSS as noted below.

WINDOW=*xx*

Setting this option changes maximum number of seconds by which the BSS database for a BSS application may be "rewound" for replay. The default value is 86400 seconds, which is 24 hours.

2.7.2 Build

To build BSS:

- Make sure that the "bp" component of ION has been built for the platform on which you plan to run BSS.
- Edit the Makefile in **ion/bss**:
- As for ici, make sure PLATFORMS is set to the name of platform on which you plan to run BSS.
- Set OPT to the directory containing the bin, lib, include, etc. directories used for building ici.
- Then:

```
cd ion/bss
make
make install
```

2.7.3 Configure

No additional configuration files are required for the operation of the BSS component of ION.

2.7.4 Run

No runtime executables are required for the operation of the BSS component of ION.

2.7.5 Test

Four test executables are provided to support testing and debugging of the BSS component of ION:

- **bssdriver** sends a stream of data to **bsscounter** for non-interactive testing.
- **bssStreamingApp** sends a stream of data to **bssrecv** for graphical, interactive testing.

For details, see the man pages for bssdriver(1), bsscounter(1), bssStreamingApp(1), and bssrecv(1) in Appendix A.

Executables (man section 1)

acsadmin
acslist
bpadmin
bpcancel
bpchat
bpclock
bpcounter
bpdriver
bpecho
bping
bplist
bprecvfile
bpsendfile
bpsink
bpsource
bpstats
bpstats2
bptrace
brsccla
brsscla
bssadmin
bssfw
dccpcli
dccpclo
dgrcla
dtn2admin
dtn2adminep
dtn2fw
hmackeys
imcadmin
imcfw
ipnadmin
ipnadminep
ipnfw
lgagent
lgsend
ltpcli
ltpclo
stcpcli
stcpclo
tcpcli
tcpclo
udpcli
udpclo
bssStreamingApp
bssrecv
bpcp
bpcpd
bputa
cfdpadmin
cfdpclock
cfdptest
dgr2file
file2dgr
file2sdr
file2sm
ionadmin
ionsecadmin
owltsim
owlttb
psmshell
psmwatch
rfxclock
sdr2file
sdrmend
sdrwatch
sm2file
smlistsh
dccplsi
dccplso
ltpadmin
ltpclock
ltpcounter
ltpdriver
ltpmeter
udplsi
udplso

Libraries (man section 3)

bp
bpextensions
bss
cfdp
dgr
ion
llcv
lyst
memmgr
platform
psm
sdr
sdrhash
sdrlist
sdrstring
sdrtable
smlist
zco
ltp

Configuration files (man section 5)

acsrc

bprc
bssrc
dtn2rc
imcrc
ipnrc
lgfile
cfdprc
ionconfig
ionrc
ionsecrc
ltprc

NAME

acsadmin – ION Aggregate Custody Signal (ACS) administration interface

SYNOPSIS

acsadmin [*commands_filename*]

DESCRIPTION

acsadmin configures aggregate custody signal behavior for the local ION node.

It operates in response to ACS configuration commands found in the file *commands_filename*, if provided; if not, **acsadmin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **acsadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *acsrc* (5).

EXIT STATUS

0 Successful completion of ACS administration.

EXAMPLES

acsadmin

Enter interactive ACS configuration command entry mode.

acsadmin host1.acs

Execute all configuration commands in *host1.acs*, then terminate immediately.

FILES

See *acsrc* (5) for details of the ACS configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the acsrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **acsadmin**. Otherwise **acsadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

acsadmin can't attach to ION.

There is no SDR data store for *acsadmin* to use. You should run *ionadmin* (1) first, to set up an SDR data store for ION.

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **acsadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *acsrc* (5) for details.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ionadmin (1), *bpadmin* (1), *acsrc* (5)

NAME

acslist – Aggregate Custody Signals (ACS) utility for checking custody IDs.

SYNOPSIS

acslist [*–s|––stdout*]

DESCRIPTION

acslist is a utility program that lists all mappings from bundle ID to custody ID currently in the local bundle agent's ACS ID database, in no specific order. A bundle ID (defined in RFC5050) is the tuple of (source EID, creation time, creation count, fragment offset, fragment length). A custody ID (defined in draft-jenkins-aggregate-custody-signals) is an integer that the local bundle agent will be able to map to a bundle ID for the purposes of aggregating and compressing custody signals.

The format for mappings is:

(ipn:13.1,333823688,95,0,0)–>(26)

While listing, **acslist** also checks the custody ID database for self-consistency, and if it detects any errors it will print a line starting with ''Mismatch:'' and describing the error.

–s|––stdout tells **acslist** to print results to stdout, rather than to the ION log.

EXIT STATUS

0 **acslist** terminated after verifying the consistency of the custody ID database.

1 **acslist** was unable to attach to the ACS database, or it detected an inconsistency.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued:

Can't attach to ACS.
: **acsadmin** has not yet initialized ACS operations.

Mismatch: (description of the mismatch)
: **acslist** detected an inconsistency in the database; this is a bug in ACS.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

acsadmin (1), *bplist* (1)

NAME

bpadmin – ION Bundle Protocol (BP) administration interface

SYNOPSIS

bpadmin [*commands_filename* | .]

DESCRIPTION

bpadmin configures, starts, manages, and stops bundle protocol operations for the local ION node.

It operates in response to BP configuration commands found in the file *commands_filename*, if provided; if not, **bpadmin** prints a simple prompt (:) so that the user may type commands directly into standard input. If *commands_filename* is a period (.), the effect is the same as if a command file containing the single command 'x' were passed to **bpadmin** — that is, the ION node's *bpclock* task, forwarder tasks, and convergence layer adapter tasks are stopped.

The format of commands for *commands_filename* can be queried from **bpadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *bprc* (5).

EXIT STATUS

0 Successful completion of BP administration.

EXAMPLES

bpadmin
: Enter interactive BP configuration command entry mode.

bpadmin host1.bp
: Execute all configuration commands in *host1.bp*, then terminate immediately.

bpadmin .
: Stop all bundle protocol operations on the local node.

FILES

See *bprc* (5) for details of the BP configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the bprc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **bpadmin**. Otherwise **bpadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

ION can't set custodian EID information.
: The *custodial_endpoint_id* specified in the BP initialization ('1') command is malformed. Remember that the format for this argument is ipn:*element_number*.0 and that the final 0 is required, as custodial/administration service is always service 0. Additional detail for this error is provided if one of the following other errors is present:

  Malformed EID.

  Malformed custodian EID.

bpadmin can't attach to ION.
: There is no SDR data store for *bpadmin* to use. You should run *ionadmin* (1) first, to set up an SDR data store for ION.

Can't open command file...
: The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **bpadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *bprc* (5) for details.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ionadmin (1), *bprc* (5), *ipnadmin* (1), *ipnrc* (5), *dtnadmin* (1), *dtnrc* (5)

NAME

bpcancel – Bundle Protocol (BP) bundle cancellation utility

SYNOPSIS

bpcancel *source_EID creation_seconds* [*creation_count* [*fragment_offset* [*fragment_length*]]]

DESCRIPTION

bpcancel locates the bundle identified by the command-line parameter values and destroys it, by simulating an immediate time-to-live expiration.

EXIT STATUS

0 **bpcancel** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

bpcancel failed finding bundle.
: No bundle identified by the command-line parameter values is present in the ''timeline'' list for the local bundle agent.

bpcancel failed destroying bundle.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

bpcancel failed.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bplist (1)

NAME

bpchat – Bundle Protocol chat test program

SYNOPSIS

bpchat *sourceEID destEID* [ct]

DESCRIPTION

bpchat uses Bundle Protocol to send input text in bundles, and display the payload of received bundles as output. It is similar to the talk utility, but operates over the Bundle Protocol. It operates like a combination of the bpsource and bpsink utilities in one program (unlike bpsource, **bpchat** emits bundles with a *sourceEID*).

If the *sourceEID* and *destEID* are both **bpchat** applications, then two users can chat with each other over the Bundle Protocol: lines that one user types on the keyboard will be transported over the network in bundles and displayed on the screen of the other user (and the reverse).

bpchat terminates upon receiving the SIGQUIT signal, i.e., ^C from the keyboard.

EXIT STATUS

0 **bpchat** has terminated normally. Any problems encountered during operation will be noted in the **ion.log** log file.

1 **bpchat** has terminated due to a BP transmit or reception failure. Details should be noted in the **ion.log** log file.

OPTIONS

[ct] If the string ‘‘ct’’ is appended as the last argument, then bundles will be sent with custody transfer requested.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpchat** are written to the ION log file *ion.log*.

Can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

Can’t open own endpoint.
: Another application has already opened *ownEndpointId*. Terminate that application and rerun.

bpchat bundle reception failed.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

No space for ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can’t create ZCO.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpchat can’t send echo bundle.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpecho (1), *bpsource* (1), *bpsink* (1), *bp* (3)

NAME

bpclock – Bundle Protocol (BP) daemon task for managing scheduled events

SYNOPSIS

bpclock

DESCRIPTION

bpclock is a background ‘‘daemon’’ task that periodically performs scheduled Bundle Protocol activities. It is spawned automatically by **bpadmin** in response to the ’s’ command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command.

Once per second, **bpclock** takes the following action:

First it (a) destroys all bundles whose TTLs have expired, (b) enqueues for re-forwarding all bundles that were expected to have been transmitted (by convergence-layer output tasks) by now but are still stuck in their assigned transmission queues, and (c) enqueues for re-forwarding all bundles for which custody has not yet been taken that were expected to have been received and acknowledged by now (as noted by invocation of the *bpMemo()* function by some convergence-layer adapter that had CL-specific insight into the appropriate interval to wait for custody acceptance).

Then **bpclock** adjusts the transmission and reception ‘‘throttles’’ that control rates of LTP transmission to and reception from neighboring nodes, in response to data rate changes as noted in the RFX database by **rfxclock**.

bpclock then checks for bundle origination activity that has been blocked due to insufficient allocated space for BP traffic in the ION data store: if space for bundle origination is now available, **bpclock** gives the bundle production throttle semaphore to unblock that activity.

Finally, **bpclock** applies rate control to all convergence-layer protocol inducts and outducts:

For each induct, **bpclock** increases the current capacity of the duct by the applicable nominal data reception rate. If the revised current capacity is greater than zero, **bpclock** gives the throttle’s semaphore to unblock data acquisition (which correspondingly reduces the current capacity of the duct) by the associated convergence layer input task.

For each outduct, **bpclock** increases the current capacity of the duct by the applicable nominal data transmission rate. If the revised current capacity is greater than zero, **bpclock** gives the throttle’s semaphore to unblock data transmission (which correspondingly reduces the current capacity of the duct) by the associated convergence layer output task.

EXIT STATUS

0 **bpclock** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **bpclock**.

1 **bpclock** was unable to attach to Bundle Protocol operations, probably because **bpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

bpclock can’t attach to BP.
bpadmin has not yet initialized BP operations.

Can’t dispatch events.
An unrecoverable database error was encountered. **bpclock** terminates.

Can't adjust throttles.

An unrecoverable database error was encountered. **bpclock** terminates.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *rfxclock* (1)

NAME

bpcounter – Bundle Protocol reception test program

SYNOPSIS

bpcounter *ownEndpointId* [*maxCount*]

DESCRIPTION

bpcounter uses Bundle Protocol to receive application data units from a remote **bpdriver** application task. When the total number of application data units it has received exceeds *maxCount*, it terminates and prints its reception count. If *maxCount* is omitted, the default limit is 2 billion application data units.

EXIT STATUS

0 **bpcounter** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpcounter** are written to the ION log file *ion.log*.

Can't attach to BP.

bpadmin has not yet initialized Bundle Protocol operations.

Can't open own endpoint.

Another application has already opened *ownEndpointId*. Terminate that application and rerun.

bpcounter bundle reception failed.

BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bpdriver* (1), *bpecho* (1), *bp* (3)

NAME

bpdriver – Bundle Protocol transmission test program

SYNOPSIS

bpdriver *nbrOfCycles ownEndpointId destinationEndpointId* [*length*] [t*TTL*]

DESCRIPTION

bpdriver uses Bundle Protocol to send *nbrOfCycles* application data units of length indicated by *length*, to a counterpart application task that has opened the BP endpoint identified by *destinationEndpointId*.

If omitted, *length* defaults to 60000.

TTL indicates the number of seconds the bundles may remain in the network, undelivered, before they are automatically destroyed. If omitted, *TTL* defaults to 300 seconds.

bpdriver normally runs in ''echo'' mode: after sending each bundle it waits for an acknowledgment bundle before sending the next one. For this purpose, the counterpart application task should be **bpecho**.

Alternatively **bpdriver** can run in ''streaming'' mode, i.e., without expecting or receiving acknowledgments. Streaming mode is enabled when *length* is specified as a negative number, in which case the additive inverse of *length* is used as the effective value of *length*. For this purpose, the counterpart application task should be **bpcounter**.

If the effective value of *length* is 1, the sizes of the transmitted service data units will be randomly selected multiples of 1024 in the range 1024 to 62464.

bpdriver normally runs with custody transfer disabled. To request custody transfer for all bundles sent by **bpdriver**, specify *nbrOfCycles* as a negative number; the additive inverse of *nbrOfCycles* will be used as its effective value in this case.

When all copies of the file have been sent, **bpdriver** prints a performance report.

EXIT STATUS

0 **bpdriver** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

FILES

The service data units transmitted by **bpdriver** are sequences of text obtained from a file in the current working directory named ''bpdriverAduFile'', which **bpdriver** creates automatically.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpdriver** are written to the ION log file *ion.log*.

Can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

Can't open own endpoint.
: Another application has already opened *ownEndpointId*. Terminate that application and rerun.

Can't create ADU file
: Operating system error. Check errtext, correct problem, and rerun.

Error writing to ADU file
: Operating system error. Check errtext, correct problem, and rerun.

bpdriver can't create file ref.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpdriver can't create ZCO.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpdriver can't send message
: Bundle Protocol service to the remote endpoint has been stopped.

bpdriver reception failed
: **bpdriver** is in "echo" mode, and Bundle Protocol delivery service has been stopped.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bpcounter* (1), *bpecho* (1), *bp* (3)

NAME

bpecho – Bundle Protocol reception test program

SYNOPSIS

bpecho *ownEndpointId*

DESCRIPTION

bpecho uses Bundle Protocol to receive application data units from a remote **bpdriver** application task. In response to each received application data unit it sends back an ''echo'' application data unit of length 2, the NULL-terminated string ''x''.

bpecho terminates upon receiving the SIGQUIT signal, i.e., ^C from the keyboard.

EXIT STATUS

0 **bpecho** has terminated normally. Any problems encountered during operation will be noted in the **ion.log** log file.

1 **bpecho** has terminated due to a BP reception failure. Details should be noted in the **ion.log** log file.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpecho** are written to the ION log file *ion.log*.

Can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

Can't open own endpoint.
: Another application has already opened *ownEndpointId*. Terminate that application and rerun.

bpecho bundle reception failed.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

No space for ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't create ZCO.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpecho can't send echo bundle.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bpdriver* (1), *bpcounter* (1), *bp* (3)

NAME

bping – Send and receive Bundle Protocol echo bundles.

SYNOPSIS

bping [**–c** *count*] [**–i** *interval*] [**–p** *priority*] [**–q** *wait*] [**–r** *flags*] [**–t** *ttl*] *srcEID destEID* [*reporttoEID*]

DESCRIPTION

bping sends bundles from *srcEID* to *destEID*. If the *destEID* echoes the bundles back (for instance, it is a bpecho endpoint), **bping** will print the round-trip time. When complete, bping will print statistics before exiting. It is very similar to ping, except it works with the bundle protocol.

bping terminates when one of the following happens: it receives the SIGINT signal (Ctrl+C), it receives responses to all of the bundles it sent, or it has sent all *count* of its bundles and waited *wait* seconds.

EXIT STATUS

These exit statuses are taken from ping.

0 **bping** has terminated normally, and received responses to all the packets it sent.

1 **bping** has terminated normally, but it did not receive responses to all the packets it sent.

2 **bping** has terminated due to an error. Details should be noted in the **ion.log** log file.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bping** are written to the ION log file *ion.log* and printed to standard error. Diagnostic messages that don't cause **bping** to terminate indicate a failure parsing an echo response bundle. This means that *destEID* isn't an echo endpoint: it's responding with some other bundle message of an unexpected format.

Can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

Can't open own endpoint.
: Another application has already opened *ownEndpointId*. Terminate that application and rerun.

bping bundle reception failed.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

No space for ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't create ZCO.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bping can't send echo bundle.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpecho (1), *bptrace* (1), *bpadmin* (1), *bp* (3), *ping* (8)

NAME

bplist – Bundle Protocol (BP) utility for listing queued bundles

SYNOPSIS

bplist

DESCRIPTION

bplist is a utility program that lists all bundles currently in the local bundle agent's ''timeline'' list, in expiration-time sequence. Identifying primary block information is printed, together with hex and ASCII dumps of the payload and all extension blocks.

EXIT STATUS

0 **bplist** terminated, for reasons noted in the **ion.log** file.

1 **bplist** was unable to attach to Bundle Protocol operations, probably because **bpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't attach to BP.
bpadmin has not yet initialized BP operations.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpclock (1)

NAME

bprecvfile – Bundle Protocol (BP) file reception utility

SYNOPSIS

bprecvfile *own_endpoint_ID* [*max_files*]

DESCRIPTION

bprecvfile is intended to serve as the counterpart to **bpsendfile**. It uses *bp_receive()* to receive bundles containing file content. The content of each bundle is simply written to a file named ''testfileN'' where N is the total number of bundles received since the program began running.

If a *max_files* value of N (where N > 0) is provided, the program will terminate automatically upon completing its Nth file reception. Otherwise it will run indefinitely; use ^C to terminate the program.

EXIT STATUS

0 **bprecvfile** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

Can't open own endpoint.
: Another BP application task currently has *own_endpoint_ID* open for bundle origination and reception. Try again after that task has terminated. If no such task exists, it may have crashed while still holding the endpoint open; the easiest workaround is to select a different source endpoint.

bprecvfile bundle reception failed.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bprecvfile: can't open test file
: File system error. **bprecvfile** terminates.

bprecvfile: can't receive bundle content.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bprecvfile: can't write to test file
: File system error. **bprecvfile** terminates.

bprecvfile cannot continue.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bprecvfile: can't handle bundle delivery.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpsendfile (1), *bp* (3)

NAME

bpsendfile – Bundle Protocol (BP) file transmission utility

SYNOPSIS

bpsendfile *own_endpoint_ID destination_endpoint_ID file_name* [*class_of_service*]

DESCRIPTION

bpsendfile uses *bp_send()* to issue a single bundle to a designated destination endpoint, containing the contents of the file identified by *file_name*, then terminates. The bundle is sent with no custody transfer requested, with TTL of 300 seconds (5 minutes). When *class_of_service* is omitted, the bundle is sent at standard priority; for details of the *class_of_service* parameter, see *bptrace* (1).

EXIT STATUS

0 **bpsendfile** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

Can't open own endpoint.
: Another BP application task currently has *own_endpoint_ID* open for bundle origination and reception. Try again after that task has terminated. If no such task exists, it may have crashed while still holding the endpoint open; the easiest workaround is to select a different source endpoint.

Can't stat the file
: Operating system error. Check errtext, correct problem, and rerun.

bpsendfile can't create file ref.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

bpsendfile can't create ZCO.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

bpsendfile can't send file in bundle.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bprecvfile (1), *bp* (3)

NAME

bpsink – Bundle Protocol reception test program

SYNOPSIS

bpsink *ownEndpointId*

DESCRIPTION

bpsink uses Bundle Protocol to receive application data units from a remote **bpsource** application task. For each application data unit it receives, it prints the ADU's length and — if length is less than 80 — its text.

bpsink terminates upon receiving the SIGQUIT signal, i.e., ^C from the keyboard.

EXIT STATUS

0 **bpsink** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpsink** are written to the ION log file *ion.log*.

Can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

Can't open own endpoint.
: Another application has already opened *ownEndpointId*. Terminate that application and rerun.

bpsink bundle reception failed.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't receive payload.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't handle delivery.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bpsource* (1), *bp* (3)

NAME

bpsource – Bundle Protocol transmission test shell

SYNOPSIS

bpsource *destinationEndpointId* ['*text*'] [–t*TTL*]

DESCRIPTION

When *text* is supplied, **bpsource** simply uses Bundle Protocol to send *text* to a counterpart **bpsink** application task that has opened the BP endpoint identified by *destinationEndpointId*, then terminates.

Otherwise, **bpsource** offers the user an interactive ''shell'' for testing Bundle Protocol data transmission. **bpsource** prints a prompt string ('': '') to stdout, accepts a string of text from stdin, uses Bundle Protocol to send the string to a counterpart **bpsink** application task that has opened the BP endpoint identified by *destinationEndpointId*, then prints another prompt string and so on. To terminate the program, enter a string consisting of a single exclamation point (!) character.

TTL indicates the number of seconds the bundles may remain in the network, undelivered, before they are automatically destroyed. If omitted, *TTL* defaults to 300 seconds.

The source endpoint ID for each bundle sent by **bpsource** is the null endpoint ID, i.e., the bundles are anonymous. All bundles are sent standard priority with no custody transfer and no status reports requested.

EXIT STATUS

0 **bpsource** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

FILES

The service data units transmitted by **bpsource** are sequences of text obtained from a file in the current working directory named ''bpsourceAduFile'', which **bpsource** creates automatically.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **bpsource** are written to the ION log file *ion.log*.

Can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

bpsource fgets failed
: Operating system error. Check errtext, correct problem, and rerun.

No space for ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't create ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpsource can't send ADU
: Bundle Protocol service to the remote endpoint has been stopped.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bpsink* (1), *bp* (3)

NAME

bpstats – Bundle Protocol (BP) processing statistics query utility

SYNOPSIS

bpstats

DESCRIPTION

bpstats simply logs messages containing the current values of all BP processing statistics accumulators, then terminates.

EXIT STATUS

0 **bpstats** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

bpstats can't attach to BP.

bpadmin has not yet initialized BP operations.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ion (3)

NAME

bpstats2 – Bundle Protocol (BP) processing statistics query utility via bundles

SYNOPSIS

bpstats2 *sourceEID* [*default destEID*] [ct]

DESCRIPTION

bpstats2 creates bundles containing the current values of all BP processing statistics accumulators. It creates these bundles when:

- an interrogation bundle is delivered to *sourceEID*: the contents of the bundle are discarded, a new statistics bundle is generated and sent to the source of the interrogation bundle. The format of the interrogation bundle is irrelevant.
- a SIGUSR1 signal is delivered to the **bpstats2** application: a new statistics bundle is generated and sent to *default destEID*.

EXIT STATUS

0 **bpstats2** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

1 **bpstats2** failed to start up or receive bundles. Diagnose the issue reported in the **ion.log** file and try again.

OPTIONS

[ct] If the string ‘‘ct’’ is appended as the last argument, then statistics bundles will be sent with custody transfer requested.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

bpstats2 can’t *bp_attach()*.
: **bpadmin** has not yet initialized BP operations.

bpstats2 can’t open own endpoint.
: Another BP application has opened that endpoint; close it and try again.

No space for ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can’t create ZCO extent.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

bpstats2 can’t send stats bundle.
: Bundle Protocol service to the remote endpoint has been stopped.

Can’t send stats: bad dest EID (dest EID)
: The destination EID printed is an invalid destination EID. The destination EID may be specified in *default destEID* or the source EID of the interrogation bundle. Ensure that *default destEID* is an EID that is valid for ION, and that the interrogator is a source EID that is also a valid destination EID. Note that ‘‘dtn:none’’ is not a valid destination EID, but is a valid source EID.

NOTES

A very simple interrogator is bpchat which can repeatedly interrogate **bpstats2** by just striking the enter key.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpstats (1), *bpchat* (1)

NAME

bptrace – Bundle Protocol (BP) network trace utility

SYNOPSIS

bptrace *own_endpoint_ID destination_endpoint_ID report−to_endpoint_ID TTL class_of_service* "*trace_text*" [*status_report_flags*]

DESCRIPTION

bptrace uses *bp_send()* to issue a single bundle to a designated destination endpoint, with status reporting options enabled as selected by the user, then terminates. The status reports returned as the bundle makes its way through the network provide a view of the operation of the network as currently configured.

TTL indicates the number of seconds the trace bundle may remain in the network, undelivered, before it is automatically destroyed.

class_of_service is *custody-requested.priority*[*.ordinal*[*.unreliable.critical*[*.flow-label*]]], where *custody-requested* must be 0 or 1 (Boolean), *priority* must be 0 (bulk) or 1 (standard) or 2 (expedited), *ordinal* must be 0–254, *unreliable* must be 0 or 1 (Boolean), *critical* must also be 0 or 1 (Boolean), and *flow-label* may be any unsigned integer. *ordinal* is ignored if *priority* is not 2. Setting *class_of_service* to ''0.2.254'' or ''1.2.254'' gives a bundle the highest possible priority. Setting *unreliable* to 1 causes BP to forego retransmission in the event of data loss, both at the BP layer and at the convergence layer. Setting *critical* to 1 causes contact graph routing to forward the bundle on all plausible routes rather than just the ''best'' route it computes; this may result in multiple copies of the bundle arriving at the destination endpoint, but when used in conjunction with priority 2.254 it ensures that the bundle will be delivered as soon as physically possible.

trace_text can be any string of ASCII text; alternatively, if we want to send a file, it can be ''@'' followed by the file name.

status_report_flags must be a sequence of status report flags, separated by commas, with no embedded whitespace. Each status report flag must be one of the following: rcv, ct, fwd, dlv, del.

EXIT STATUS

0 **bptrace** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

bptrace can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

bptrace can't open own endpoint.
: Another BP application task currently has *own_endpoint_ID* open for bundle origination and reception. Try again after that task has terminated. If no such task exists, it may have crashed while still holding the endpoint open; the easiest workaround is to select a different source endpoint.

No space for bptrace text.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

bptrace can't create ZCO.
: Probably an unrecoverable database error, in which case the local ION node must be terminated and re-initialized.

bptrace can't send message.
: BP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bp (3)

NAME

brsccla – BRSC–based BP convergence layer adapter (input and output) task

SYNOPSIS

brsccla *server_hostname*[:*server_port_nbr*]*_own_node_nbr*

DESCRIPTION

BRSC is the ‘‘client’’ side of the Bundle Relay Service (BRS) convergence layer protocol for BP. It is complemented by BRSS, the ‘‘server’’ side of the BRS convergence layer protocol for BP. BRS clients send bundles directly only to the server, regardless of their final destinations, and the server forwards them to other clients as necessary.

brsccla is a background ‘‘daemon’’ task comprising three threads: one that connects to the BRS server, spawns the other threads, and then handles BRSC protocol output by transmitting bundles over the connected socket to the BRS server; one that simply sends periodic ‘‘keepalive’’ messages over the connected socket to the server (to assure that local inactivity doesn’t cause the connection to be lost); and one that handles BRSC protocol input from the connected server.

The output thread connects to the server’s TCP socket at *server_hostname* and *server_port_nbr*, sends over the connected socket the client’s *own_node_nbr* (in SDNV representation) followed by a 32–bit time tag and a 160–bit HMAC–SHA1 digest of that time tag, to authenticate itself; checks the authenticity of the 160–bit countersign returned by the server; spawns the keepalive and receiver threads; and then begins extracting bundles from the queues of bundles ready for transmission via BRSC and transmitting those bundles over the connected socket to the server. Each transmitted bundle is preceded by its length, a 32–bit unsigned integer in network byte order. The default value for *server_port_nbr*, if omitted, is 80.

The reception thread receives bundles over the connected socket and passes them to the bundle protocol agent on the local ION node. Each bundle received on the connection is preceded by its length, a 32–bit unsigned integer in network byte order.

The keepalive thread simply sends a ‘‘bundle length’’ value of zero (a 32–bit unsigned integer in network byte order) to the server once every 15 seconds.

Note that **brsccla** is not a ‘‘promiscuous’’ convergence layer daemon: it can transmit bundles only to the BRS server to which it is connected, so scheme configuration directives that cite this outduct need only provide the protocol name and the BRSC outduct name as specified on the command line when **brsccla** is started.

brsccla is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **brsccla** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the BRSC convergence layer protocol.

EXIT STATUS

0 **brsccla** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart the BRSC protocol.

1 **brsccla** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart the BRSC protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

brsccla can’t attach to BP.

bpadmin has not yet initialized Bundle Protocol operations.

No such brsc induct.
: No BRSC induct with duct name matching *server_hostname*, *own_node_nbr*, and *server_port_nbr* has been added to the BP database. Use **bpadmin** to stop the BRSC convergence-layer protocol, add the induct, and then restart the BRSC protocol.

CLI task is already started for this duct.
: Redundant initiation of **brsccla**.

No such brsc outduct.
: No BRSC outduct with duct name matching *server_hostname*, *own_node_nbr*, and *server_port_nbr* has been added to the BP database. Use **bpadmin** to stop the BRSC convergence-layer protocol, add the outduct, and then restart the BRSC protocol.

Can't connect to server.
: Operating system error. Check errtext, correct problem, and restart BRSC.

Can't register with server.
: Configuration error. Authentication has failed, probably because (a) the client and server are using different HMAC/SHA1 keys or (b) the clocks of the client and server differ by more than 5 seconds. Update security policy database(s), as necessary, and assure that the clocks are synchronized.

brsccla can't create receiver thread
: Operating system error. Check errtext, correct problem, and restart BRSC.

brsccla can't create keepalive thread
: Operating system error. Check errtext, correct problem, and restart BRSC.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *brsscla* (1)

NAME

brsscla – BRSS–based BP convergence layer adapter (input and output) task

SYNOPSIS

brsscla *local_hostname*[:*local_port_nbr*][*first_duct_nbr_in_scope*[*last_duct_nbr_in_scope*]]

DESCRIPTION

BRSS is the ‘‘server’’ side of the Bundle Relay Service (BRS) convergence layer protocol for BP. It is complemented by BRSC, the ‘‘client’’ side of the BRS convergence layer protocol for BP.

brsscla is a background ‘‘daemon’’ task that spawns two plus N threads: one that handles BRSS client connections and spawns sockets for continued data interchange with connected clients; one that handles BRSS protocol output by transmitting over those spawned sockets to the associated clients; and one input thread for each spawned socket, to handle BRSS protocol input from the associated connected client.

The connection thread simply accepts connections on a TCP socket bound to *local_hostname* and *local_port_nbr* and spawns reception threads. The default value for *local_port_nbr*, if omitted, is 80.

Each reception thread receives over the socket connection the node number of the connecting client (in SDNV representation), followed by a 32–bit time tag and a 160–bit HMAC–SHA1 digest of that time tag. The node number must be in the range *first_duct_nbr_in_scope* through *last_duct_nbr_in_scope* inclusive; when omitted, *first_duct_nbr_in_scope* defaults to 1 and *last_duct_nbr_in_scope* defaults to *first_duct_nbr_in_scope* plus 255. The receiving thread also checks the time tag, requiring that it differ from the current time by no more than BRSTERM (default value 5) seconds. It then recomputes the digest value using the HMAC–SHA1 key named "*node_number*.brs" as recorded in the ION security database (see *ionsecrc* (5)), requiring that the supplied and computed digests be identical. If all registration conditions are met, the receiving thread sends the client a countersign — a similarly computed HMAC–SHA1 digest, for the time tag that is 1 second later than the provided time tag — to assure the client of its own authenticity, then commences receiving bundles over the connected socket. Each bundle received on the connection is preceded by its length, a 32–bit unsigned integer in network byte order. The received bundles are passed to the bundle protocol agent on the local ION node.

The output thread extracts bundles from the queues of bundles ready for transmission via BRSS to remote bundle protocol agents, finds the connected clients whose node numbers match the proximate receiver node numbers assigned to the bundles by the routing daemons that enqueued them, and transmits the bundles over the sockets to those clients. Each transmitted bundle is preceded by its length, a 32–bit unsigned integer in network byte order.

Note that **brsscla** is a ‘‘promiscuous’’ convergence layer daemon, able to transmit bundles to any BRSS destination induct for which it has received a connection. Its sole outduct’s name is the name of the corresponding induct, rather than the induct name of any single BRSS destination induct to which the outduct might be dedicated, so scheme configuration directives that cite this outduct must provide destination induct IDs. For the BRS convergence-layer protocol, destination induct IDs are simply the node numbers of connected clients.

brsscla is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **brsscla** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the BRSS convergence layer protocol.

EXIT STATUS

0 **brsscla** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart the BRSS protocol.

1 **brsscla** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart the BRSS protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

brsscla can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such brss induct.
: No BRSS induct with duct name matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the BRSS convergence-layer protocol, add the induct, and then restart the BRSS protocol.

CLI task is already started for this duct.
: Redundant initiation of **brsscla**.

No such brss outduct.
: No BRSS outduct with duct name matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the BRSS convergence-layer protocol, add the outduct, and then restart the BRSS protocol.

Can't get IP address for host
: Operating system error. Check errtext, correct problem, and restart BRSS.

Can't open TCP socket
: Operating system error — unable to open TCP socket for accepting connections. Check errtext, correct problem, and restart BRSS.

Can't initialize socket (note: must be root for port 80)
: Operating system error. Check errtext, correct problem, and restart BRSS.

brsscla can't create sender thread
: Operating system error. Check errtext, correct problem, and restart BRSS.

brsscla can't create access thread
: Operating system error. Check errtext, correct problem, and restart BRSS.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *brsccla* (1)

NAME

bssadmin – Interplanetary Internet (IPN) scheme administration interface for use with Bundle Streaming Service

SYNOPSIS

bssadmin [*commands_filename*]

DESCRIPTION

bssadmin is a BSS variant of *ipnadmin* (1) that configures the local ION node's routing of bundles to endpoints whose IDs conform to the *ipn* endpoint ID scheme. *ipn* is a CBHE-conformant scheme; that is, every endpoint ID in the *ipn* scheme is a string of the form "ipn:*element_number.service_number*" where *element_number* is a CBHE ''node number'' and *service_number* identifies a specific application processing point.

bssadmin operates in response to IPN scheme configuration commands found in the file *commands_filename*, if provided; if not, **bssadmin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **bssadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *bssrc* (5).

EXIT STATUS

0 Successful completion of IPN scheme administration.

1 Unsuccessful completion of IPN scheme administration, due to inability to attach to the Bundle Protocol system or to initialize the IPN scheme.

EXAMPLES

bssadmin

Enter interactive IPN scheme configuration command entry mode.

bssadmin host1.bssrc

Execute all configuration commands in *host1.bssrc*, then terminate immediately.

FILES

See *bssrc* (5) for details of the BSS configuration commands for the IPN scheme.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the bssrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **bssadmin**. Otherwise **bssadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

bssadmin can't attach to BP.

Bundle Protocol has not been initialized on this computer. You need to run *bpadmin* (1) first.

bssadmin can't initialize routing database.

There is no SDR data store for *dtnadmin* to use. Please run *ionadmin* (1) to start the local ION node.

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **bssadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *bssrc* (5) for details.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bssrc (5)

NAME

bssfw – bundle route computation task for the IPN scheme, adapted for Bundle Streaming Service

SYNOPSIS

bssfw

DESCRIPTION

bssfw is a background ''daemon'' task that pops bundles from the queue of bundle destined for IPN-scheme endpoints, computes proximate destinations for those bundles, and appends those bundles to the appropriate queues of bundles pending transmission to those computed proximate destinations.

For each possible proximate destination (that is, neighboring node) there is a separate queue for each possible level of bundle priority: 0, 1, 2. Each outbound bundle is appended to the queue matching the bundle's designated priority.

Proximate destination computation is affected by static and default routes as configured by *bssadmin* (1) and by contact graphs as managed by *ionadmin* (1) and *rfxclock* (1). **bssfw** differs from **ipnfw** in this way: a bundle that is destined for an endpoint associated with a BSS application (as registered in an ''entry'' passed to *bssadmin* (1) in a *bssrc* (5) command) is forwarded via a duct identified in a separately defined real-time or playback duct expression, rather than the standard duct that is used for non-BSS traffic.

bssfw is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **bssfw** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the IPN scheme.

EXIT STATUS

0 **bssfw** terminated, for reasons noted in the **ion.log** log file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **bssfw**.

1 **bssfw** could not commence operations, for reasons noted in the **ion.log** log file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **bssfw**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

bssfw can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

bssfw can't load routing database.
: **bssadmin** has not yet initialized the IPN scheme.

Can't create lists for route computation.
: An unrecoverable database error was encountered. **bssfw** terminates.

'bss' scheme is unknown.
: The IPN scheme was not added when **bpadmin** initialized BP operations. Use **bpadmin** to add and start the scheme.

Can't take forwarder semaphore.
: ION system error. **bssfw** terminates.

Can't exclude sender from routes.
: An unrecoverable database error was encountered. **bssfw** terminates.

Can't enqueue bundle.
: An unrecoverable database error was encountered. **bssfw** terminates.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bssadmin* (1), *bprc* (5), *bssrc* (5).

NAME

dccpcli – DCCP–based BP convergence layer input task

SYNOPSIS

dccpcli *local_hostname*[:*local_port_nbr*]

DESCRIPTION

dccpcli is a background ‘‘daemon’’ task that receives DCCP datagrams via a DCCP socket bound to *local_hostname* and *local_port_nbr*, extracts bundles from those datagrams, and passes them to the bundle protocol agent on the local ION node.

If not specified, port number defaults to 4556.

Note that **dccpcli** has no fragmentation support at all. Therefore, the largest bundle that can be sent via this convergence layer is limited to just under the link's MTU (typically 1500 bytes).

The convergence layer input task is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of the Bundle Protocol; the text of the command that is used to spawn the task must be provided at the time the ‘‘dccp’’ convergence layer protocol is added to the BP database. The convergence layer input task is terminated by **bpadmin** in response to an 'x' (STOP) command. **dccpcli** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the DCCP convergence layer protocol.

EXIT STATUS

0 **dccpcli** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **dccpcli**.

1 **dccpcli** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **dccpcli**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dccpcli can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such dccp duct.
: No DCCP induct matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the DCCP convergence-layer protocol, add the induct, and then restart the DCCP protocol.

CLI task is already started for this duct.
: Redundant initiation of **dccpcli**.

dccpcli can't get IP address for host.
: Operating system error. Check errtext, correct problem, and restart **dccpcli**.

CLI can't open DCCP socket. This probably means DCCP is not supported on your system.
: Operating system error. This probably means that you are not using an operating system that supports DCCP. Make sure that you are using a current Linux kernel and that the DCCP modules are being compiled. Check errtext, correct problem, and restart **dccpcli**.

CLI can't initialize socket.
: Operating system error. Check errtext, correct problem, and restart **dccpcli**.

dccpcli can't get acquisition work area.
: ION system error. Check errtext, correct problem, and restart **dccpcli**.

dccpcli can't create new thread.

Operating system error. Check errtext, correct problem, and restart **dccpcli**.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *dccpclo* (1)

NAME

dccpclo – DCCP–based BP convergence layer output task

SYNOPSIS

dccpclo *remote_hostname*[:*remote_port_nbr*]

DESCRIPTION

dccpclo is a background ''daemon'' task that connects to a remote node's DCCP socket at *remote_hostname* and *remote_port_nbr*. It then begins extracting bundles from the queues of bundles ready for transmission via DCCP to this remote bundle protocol agent and transmitting those bundles as DCCP datagrams to the remote host.

If not specified, *remote_port_nbr* defaults to 4556.

Note that **dccpclo** is not a ''promiscuous'' convergence layer daemon: it can transmit bundles only to the node to which it is connected, so scheme configuration directives that cite this outduct need only provide the protocol name and the outduct name as specified on the command line when **dccpclo** is started.

Note also that **dccpclo** has no fragmentation support at all. Therefore, the largest bundle that can be sent via this convergence layer is limited to just under the link's MTU (typically 1500 bytes).

dccpclo is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **dccpclo** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the DCCP convergence layer protocol.

EXIT STATUS

0 **dccpclo** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **dccpclo**.

1 **dccpclo** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **dccpclo**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dccpclo can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No memory for DCCP buffer in dccpclo.
: ION system error. Check errtext, correct problem, and restart **dccpclo**.

No such dccp duct.
: No DCCP outduct matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the DCCP convergence-layer protocol, add the outduct, and then restart **dccpclo**.

CLO task is already started for this duct.
: Redundant initiation of **dccpclo**.

dccpclo can't get IP address for host.
: Operating system error. Check errtext, correct problem, and restart **dccpclo**.

dccpclo can't create thread.
: Operating system error. Check errtext, correct problem, and restart **dccpclo**.

CLO can't open DCCP socket. This probably means DCCP is not supported on your system.
: Operating system error. This probably means that you are not using an operating system that supports DCCP. Make sure that you are using a current Linux kernel and that the DCCP modules are being

compiled. Check errtext, correct problem, and restart **dccpclo**.

CLO can't initialize socket.
Operating system error. Check errtext, correct problem, and restart **dccpclo**.

CLO *send()* error on socket.
Operating system error. Check errtext, correct problem, and restart **dccpclo**.

Bundle is too big for DCCP CLO.
Configuration error: bundles that are too large for DCCP transmission (i.e., larger than the MTU of the link or 65535 bytes — whichever is smaller) are being enqueued for **dccpclo**. Change routing or use smaller bundles.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *dccpcli* (1)

NAME

dgrcla – DGR−based BP convergence layer adapter (input and output) task

SYNOPSIS

dgrcla *local_hostname*[:*local_port_nbr*]

DESCRIPTION

dgrcla is a background ‘‘daemon’’ task that spawns two threads, one that handles DGR convergence layer protocol input and a second that handles DGR convergence layer protocol output.

The input thread receives DGR messages via a UDP socket bound to *local_hostname* and *local_port_nbr*, extracts bundles from those messages, and passes them to the bundle protocol agent on the local ION node. (*local_port_nbr* defaults to 1113 if not specified.)

The output thread extracts bundles from the queues of bundles ready for transmission via DGR to remote bundle protocol agents, encapsulates them in DGR messages, and sends those messages to the appropriate remote UDP sockets as indicated by the host names and UDP port numbers (destination induct names) associated with the bundles by the routing daemons that enqueued them.

Note that **dgrcla** is a ‘‘promiscuous’’ convergence layer daemon, able to transmit bundles to any DGR destination induct. Its duct name is the name of the corresponding induct, rather than the induct name of any single DGR destination induct to which it might be dedicated, so scheme configuration directives that cite this outduct must provide destination induct IDs. For the DGR convergence-layer protocol, destination induct IDs are identical to induct names, i.e., they are of the form *local_hostname*[:*local_port_nbr*].

dgrcla is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **dgrcla** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the DGR convergence layer protocol.

EXIT STATUS

0 **dgrcla** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **dgrcla**.

1 **dgrcla** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **dgrcla**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dgrcla can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such dgr induct.
: No DGR induct with duct name matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the DGR convergence-layer protocol, add the induct, and then restart the DGR protocol.

CLI task is already started for this engine.
: Redundant initiation of **dgrcla**.

No such dgr induct.
: No DGR outduct with duct name matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the DGR convergence-layer protocol, add the outduct, and then restart the DGR protocol.

Can't get IP address for host
: Operating system error. Check errtext, correct problem, and restart DGR.

dgrcla can't open DGR service access point.
: DGR system error. Check prior messages in **ion.log** log file, correct problem, and then stop and restart the DGR protocol.

dgrcla can't create sender thread
: Operating system error. Check errtext, correct problem, and restart DGR.

dgrcla can't create receiver thread
: Operating system error. Check errtext, correct problem, and restart DGR.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5)

NAME

dtn2admin – baseline "dtn" scheme administration interface

SYNOPSIS

dtn2admin [*commands_filename*]

DESCRIPTION

dtn2admin configures the local ION node's routing of bundles to endpoints whose IDs conform to the *dtn* endpoint ID scheme. *dtn* is a non-CBHE-conformant scheme. The structure of *dtn* endpoint IDs remains somewhat in flux at the time of this writing, but endpoint IDs in the *dtn* scheme historically have been strings of the form "dtn://*node_name*[/*demux_token*]", where *node_name* normally identifies a computer somewhere on the network and *demux_token* normally identifies a specific application processing point. Although the *dtn* endpoint ID scheme imposes more transmission overhead than the *ipn* scheme, ION provides support for *dtn* endpoint IDs to enable interoperation with other implementations of Bundle Protocol.

dtn2admin operates in response to ''dtn'' scheme configuration commands found in the file *commands_filename*, if provided; if not, **dtn2admin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **dtn2admin** with the 'h' or '?' commands at the prompt. The commands are documented in *dtn2rc* (5).

EXIT STATUS

0 Successful completion of ''dtn'' scheme administration.

1 Unsuccessful completion of ''dtn'' scheme administration, due to inability to attach to the Bundle Protocol system or to initialize the ''dtn'' scheme.

EXAMPLES

dtn2admin

Enter interactive ''dtn'' scheme configuration command entry mode.

dtn2admin host1.dtn2rc

Execute all configuration commands in *host1.dtn2rc*, then terminate immediately.

FILES

See *dtn2rc* (5) for details of the DTN scheme configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the dtn2rc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **dtn2admin**. Otherwise **dtn2admin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

dtn2admin can't attach to BP.

Bundle Protocol has not been initialized on this computer. You need to run *bpadmin* (1) first.

dtn2admin can't initialize routing database.

There is no SDR data store for *dtn2admin* to use. Please run *ionadmin* (1) to start the local ION node.

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **dtn2admin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *dtn2rc* (5) for details.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

dtn2rc (5)

NAME

dtn2adminep – administrative endpoint task for the "dtn" scheme

SYNOPSIS

dtn2adminep

DESCRIPTION

dtn2adminep is a background ‘‘daemon’’ task that receives and processes administrative bundles (all custody signals and, nominally, all bundle status reports) that are sent to the ‘‘dtn’’–scheme administrative endpoint on the local ION node, if and only if such an endpoint was established by **bpadmin**. It is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **dtn2adminep** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the ‘‘dtn’’ scheme.

dtn2adminep responds to custody signals as specified in the Bundle Protocol specification, RFC 5050. It responds to bundle status reports by logging ASCII text messages describing the reported activity.

EXIT STATUS

0 **dtn2adminep** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **dtn2adminep**.

1 **dtn2adminep** was unable to attach to Bundle Protocol operations or was unable to load the ‘‘dtn’’ scheme database, probably because **bpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dtn2adminep can’t attach to BP.
: **bpadmin** has not yet initialized BP operations.

dtn2adminep can’t load routing database.
: **dtn2admin** has not yet initialized the ‘‘dtn’’ scheme.

dtn2adminep can’t get admin EID.
: **dtn2admin** has not yet initialized the ‘‘dtn’’ scheme.

dtn2adminep crashed.
: An unrecoverable database error was encountered. **dtn2adminep** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *dtn2admin* (1).

NAME

dtn2fw – bundle route computation task for the "dtn" scheme

SYNOPSIS

dtn2fw

DESCRIPTION

dtn2fw is a background ‘‘daemon’’ task that pops bundles from the queue of bundle destined for ‘‘dtn’’–scheme endpoints, computes proximate destinations for those bundles, and appends those bundles to the appropriate queues of bundles pending transmission to those computed proximate destinations.

For each possible proximate destination (that is, neighboring node) there is a separate queue for each possible level of bundle priority: 0, 1, 2. Each outbound bundle is appended to the queue matching the bundle's designated priority.

Proximate destination computation is affected by static routes as configured by *dtn2admin* (1).

dtn2fw is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **dtn2fw** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the ‘‘dtn’’ scheme.

EXIT STATUS

0 **dtn2fw** terminated, for reasons noted in the **ion.log** log file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **dtn2fw**.

1 **dtn2fw** could not commence operations, for reasons noted in the **ion.log** log file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **dtn2fw**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dtn2fw can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

dtn2fw can't load routing database.
: **dtn2admin** has not yet initialized the ‘‘dtn’’ scheme.

Can't create lists for route computation.
: An unrecoverable database error was encountered. **dtn2fw** terminates.

'dtn' scheme is unknown.
: The ‘‘dtn’’ scheme was not added when **bpadmin** initialized BP operations. Use **bpadmin** to add and start the scheme.

Can't take forwarder semaphore.
: ION system error. **dtn2fw** terminates.

Can't enqueue bundle.
: An unrecoverable database error was encountered. **dtn2fw** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *dtn2admin* (1), *bprc* (5), *dtn2rc* (5).

NAME

hmackeys – utility program for generating good HMAC–SHA1 keys

SYNOPSIS

hmackeys [*keynames_filename*]

DESCRIPTION

hmackeys writes files containing randomized 160–bit key values suitable for use by HMAC–SHA1 in support of Bundle Authentication Block processing, Bundle Relay Service connections, or other functions for which symmetric hash computation is applicable. One file is written for each key name presented to *hmackeys*; the content of each file is 20 consecutive randomly selected 8–bit integer values, and the name given to each file is simply "*keyname*.hmk".

hmackeys operates in response to the key names found in the file *keynames_filename*, one name per file text line, if provided; if not, **hmackeys** prints a simple prompt (:) so that the user may type key names directly into standard input.

When the program is run in interactive mode, either enter 'q' or press ^C to terminate.

EXIT STATUS

0 Completion of key generation.

EXAMPLES

hmackeys
: Enter interactive HMAC/SHA1 key generation mode.

hmackeys host1.keynames
: Create a key file for each key name in *host1.keynames*, then terminate immediately.

FILES

No other files are used in the operation of *hmackeys*.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the logfile ion.log:

Can't open keynames file...
: The *keynames_filename* specified in the command line doesn't exist.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

brsscla (1), *ionsecadmin* (1)

NAME

imcadmin – Interplanetary Multicast (IMC) scheme administration interface

SYNOPSIS

imcadmin [*commands_filename*]

DESCRIPTION

imcadmin configures the local ION node's routing of bundles to endpoints whose IDs conform to the *imc* endpoint ID scheme. *imc* is a CBHE-conformant scheme; that is, every endpoint ID in the *imc* scheme is a string of the form "imc:*group_number*.*service_number*" where *group_number* (an IMC multicast group number) serves as a CBHE ''node number'' and *service_number* identifies a specific application processing point.

imcadmin operates in response to IMC scheme configuration commands found in the file *commands_filename*, if provided; if not, **imcadmin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **imcadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *imcrc* (5).

EXIT STATUS

0 Successful completion of IMC scheme administration.
1 Unsuccessful completion of IMC scheme administration, due to inability to attach to the Bundle Protocol system or to initialize the IMC scheme.

EXAMPLES

imcadmin
: Enter interactive IMC scheme configuration command entry mode.

imcadmin host1.imcrc
: Execute all configuration commands in *host1.ipnrc*, then terminate immediately.

FILES

See *imcrc* (5) for details of the IMC scheme configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the ipnrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **imcadmin**. Otherwise **imcadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

imcadmin can't attach to BP.
: Bundle Protocol has not been initialized on this computer. You need to run *bpadmin* (1) first.

imcadmin can't initialize routing database.
: There is no SDR data store for *imcadmin* to use. Please run *ionadmin* (1) to start the local ION node.

Can't open command file...
: The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **imcadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *imcrc* (5) for details.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

imcrc (5)

NAME

imcfw – bundle route computation task for the IMC scheme

SYNOPSIS

imcfw

DESCRIPTION

imcfw is a background ‘‘daemon’’ task that pops bundles from the queue of bundle destined for IMC-scheme (Interplanetary Multicast) endpoints, determines which ‘‘relatives’’ on the IMC multicast tree to forward the bundles to, and appends those bundles to the appropriate queues of bundles pending transmission to those proximate destinations.

For each possible proximate destination (that is, neighboring node) there is a separate queue for each possible level of bundle priority: 0, 1, 2. Each outbound bundle is appended to the queue matching the bundle's designated priority.

Proximate destination computation is determined by multicast group membership as resulting from nodes' registration in multicast endpoints, governed by multicast tree structure as configured by *imcadmin* (1).

imcfw is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **imcfw** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the IMC scheme.

EXIT STATUS

0 **imcfw** terminated, for reasons noted in the **ion.log** log file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **imcfw**.

1 **imcfw** could not commence operations, for reasons noted in the **ion.log** log file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **imcfw**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

imcfw can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

imcfw can't load routing database.
: **ipnadmin** has not yet initialized the IPN scheme.

Can't create lists for route computation.
: An unrecoverable database error was encountered. **imcfw** terminates.

'imc' scheme is unknown.
: The IMC scheme was not added when **bpadmin** initialized BP operations. Use **bpadmin** to add and start the scheme.

Can't take forwarder semaphore.
: ION system error. **imcfw** terminates.

Can't exclude sender from routes.
: An unrecoverable database error was encountered. **imcfw** terminates.

Can't enqueue bundle.
: An unrecoverable database error was encountered. **imcfw** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *imcadmin* (1), *bprc* (5), *imcrc* (5)

NAME

ipnadmin – Interplanetary Internet (IPN) scheme administration interface

SYNOPSIS

ipnadmin [*commands_filename*]

DESCRIPTION

ipnadmin configures the local ION node's routing of bundles to endpoints whose IDs conform to the *ipn* endpoint ID scheme. *ipn* is a CBHE-conformant scheme; that is, every endpoint ID in the *ipn* scheme is a string of the form "ipn:*node_number.service_number*" where *node_number* is a CBHE ''node number'' and *service_number* identifies a specific application processing point.

ipnadmin operates in response to IPN scheme configuration commands found in the file *commands_filename*, if provided; if not, **ipnadmin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **ipnadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *ipnrc* (5).

EXIT STATUS

0 Successful completion of IPN scheme administration.

1 Unsuccessful completion of IPN scheme administration, due to inability to attach to the Bundle Protocol system or to initialize the IPN scheme.

EXAMPLES

ipnadmin

Enter interactive IPN scheme configuration command entry mode.

ipnadmin host1.ipnrc

Execute all configuration commands in *host1.ipnrc*, then terminate immediately.

FILES

See *ipnrc* (5) for details of the IPN scheme configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the ipnrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **ipnadmin**. Otherwise **ipnadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

ipnadmin can't attach to BP.

Bundle Protocol has not been initialized on this computer. You need to run *bpadmin* (1) first.

ipnadmin can't initialize routing database.

There is no SDR data store for *ipnadmin* to use. Please run *ionadmin* (1) to start the local ION node.

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **ipnadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *ipnrc* (5) for details.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ipnrc (5)

NAME

ipnadminep – administrative endpoint task for the IPN scheme

SYNOPSIS

ipnadminep

DESCRIPTION

ipnadminep is a background ‘‘daemon’’ task that receives and processes administrative bundles (all custody signals and, nominally, all bundle status reports) that are sent to the IPN-scheme administrative endpoint on the local ION node, if and only if such an endpoint was established by **bpadmin**. It is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **ipnadminep** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the IPN scheme.

ipnadminep responds to custody signals as specified in the Bundle Protocol specification, RFC 5050. It responds to bundle status reports by logging ASCII text messages describing the reported activity.

EXIT STATUS

0 **ipnadminep** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **ipnadminep**.

1 **ipnadminep** was unable to attach to Bundle Protocol operations or was unable to load the IPN scheme database, probably because **bpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ipnadminep can’t attach to BP.
bpadmin has not yet initialized BP operations.

ipnadminep can’t load routing database.
ipnadmin has not yet initialized the IPN scheme.

ipnadminep crashed.
An unrecoverable database error was encountered. **ipnadminep** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *ipnadmin* (1), *bprc* (5).

NAME

ipnfw – bundle route computation task for the IPN scheme

SYNOPSIS

ipnfw

DESCRIPTION

ipnfw is a background "daemon" task that pops bundles from the queue of bundle destined for IPN-scheme endpoints, computes proximate destinations for those bundles, and appends those bundles to the appropriate queues of bundles pending transmission to those computed proximate destinations.

For each possible proximate destination (that is, neighboring node) there is a separate queue for each possible level of bundle priority: 0, 1, 2. Each outbound bundle is appended to the queue matching the bundle's designated priority.

Proximate destination computation is affected by static and default routes as configured by *ipnadmin* (1) and by contact graphs as managed by *ionadmin* (1) and *rfxclock* (1).

ipnfw is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of Bundle Protocol on the local ION node, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **ipnfw** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the IPN scheme.

EXIT STATUS

0 **ipnfw** terminated, for reasons noted in the **ion.log** log file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **ipnfw**.

1 **ipnfw** could not commence operations, for reasons noted in the **ion.log** log file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **ipnfw**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ipnfw can't attach to BP.
: **bpadmin** has not yet initialized BP operations.

ipnfw can't load routing database.
: **ipnadmin** has not yet initialized the IPN scheme.

Can't create lists for route computation.
: An unrecoverable database error was encountered. **ipnfw** terminates.

'ipn' scheme is unknown.
: The IPN scheme was not added when **bpadmin** initialized BP operations. Use **bpadmin** to add and start the scheme.

Can't take forwarder semaphore.
: ION system error. **ipnfw** terminates.

Can't exclude sender from routes.
: An unrecoverable database error was encountered. **ipnfw** terminates.

Can't enqueue bundle.
: An unrecoverable database error was encountered. **ipnfw** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *ipnadmin* (1), *bprc* (5), *ipnrc* (5)

NAME

lgagent – ION Load/Go remote agent program

SYNOPSIS

lgagent *own_endpoint_ID*

DESCRIPTION

ION Load/Go is a system for management of an ION-based network, enabling the execution of ION administrative programs at remote nodes. The system comprises two programs, **lgsend** and **lgagent**.

The **lgagent** task on a given node opens the indicated ION endpoint for bundle reception, receives the extracted payloads of Load/Go bundles sent to it by **lgsend** as run on one or more remote nodes, and processes those payloads, which are the text of Load/Go source files.

Load/Go source file content is limited to newline-terminated lines of ASCII characters. More specifically, the text of any Load/Go source file is a sequence of *line sets* of two types: *file capsules* and *directives*. Any Load/Go source file may contain any number of file capsules and any number of directives, freely intermingled in any order, but the typical structure of a Load/Go source file is simply a single file capsule followed by a single directive.

When **lgagent** identifies a file capsule, it copies all of the capsule's text lines to a new file that it creates in the current working directory. When **lgagent** identifies a directive, it executes the directive by passing the text of the directive to the *pseudoshell()* function (see *platform* (3)). **lgagent** processes the line sets of a Load/Go source file in the order in which they appear in the file, so the text of a directive may reference a file that was created as the result of processing a prior file capsule in the same source file.

EXIT STATUS

0 Load/Go remote agent processing has terminated.

FILES

lgfile contains the Load/Go file capsules and directives that are to be processed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

lgagent: can't attach to BP.
: Bundle Protocol is not running on this computer. Run *bpadmin* (1) to start BP.

lgagent: can't open own endpoint.
: *own_endpoint_ID* is not a declared endpoint on the local ION node. Run *bpadmin* (1) to add it.

lgagent: bundle reception failed.
: ION system problem. Investigate and correct before restarting.

lgagent cannot continue.
: lgagent processing problem. See earlier diagnostic messages for details. Investigate and correct before restarting.

lgagent: no space for bundle content.
: ION system problem: have exhausted available SDR data store reserves.

lgagent: can't receive bundle content.
: ION system problem: have exhausted available SDR data store reserves.

lgagent: can't handle bundle delivery.
: ION system problem. Investigate and correct before restarting.

lgagent: pseudoshell failed.
: Error in directive line, usually an attempt to execute a non-existent administration program (e.g., a misspelled program name). Terminates processing of source file content.

A variety of other diagnostics noting source file parsing problems may also be reported. These errors are

non-fatal but they terminate the processing of the source file content from the most recently received bundle.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

lgsend (1), *lgfile* (5)

NAME

lgsend – ION Load/Go command program

SYNOPSIS

lgsend *command_file_name own_endpoint_ID destination_endpoint_ID*

DESCRIPTION

ION Load/Go is a system for management of an ION-based network, enabling the execution of ION administrative programs at remote nodes. The system comprises two programs, **lgsend** and **lgagent**.

The **lgsend** program reads a Load/Go source file from a local file system, encapsulates the text of that source file in a bundle, and sends the bundle to an **lgagent** task that is waiting for data at a designated DTN endpoint on the remote node.

To do so, it first reads all lines of the Load/Go source file identified by *command_file_name* into a temporary buffer in ION's SDR data store, concatenating the lines of the file and retaining all newline characters. Then it invokes the *bp_send()* function to create and send a bundle whose payload is this temporary buffer, whose destination is *destination_endpoint_ID*, and whose source endpoint ID is *own_endpoint_ID*. Then it terminates.

EXIT STATUS

0 Load/Go file transmission succeeded.

1 Load/Go file transmission failed. Examine **ion.log** to determine the cause of the failure, then re-run.

FILES

lgfile contains the Load/Go file capsules and directive that are to be sent to the remote node.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

lgsend: can't attach to BP.
: Bundle Protocol is not running on this computer. Run *bpadmin* (1) to start BP.

lgsend: can't open own endpoint.
: *own_endpoint_ID* is not a declared endpoint on the local ION node. Run *bpadmin* (1) to add it.

lgsend: can't open file of LG commands: *error description*
: *command_file_name* doesn't identify a file that can be opened. Correct spelling of file name or file's access permissions.

lgsend: can't get size of LG command file: *error description*
: Operating system problem. Investigate and correct before rerunning.

lgsend: LG cmd file size > 64000.
: Load/Go command file is too large. Split it into multiple files if possible.

lgsend: no space for application data unit.
: ION system problem: have exhausted available SDR data store reserves.

lgsend: fgets failed: *error description*
: Operating system problem. Investigate and correct before rerunning.

lgsend: can't create application data unit.
: ION system problem: have exhausted available SDR data store reserves.

lgsend: can't send bundle.
: ION system problem. Investigate and correct before rerunning.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

lgagent (1), *lgfile* (5)

NAME

ltpcli – LTP–based BP convergence layer input task

SYNOPSIS

ltpcli *local_node_nbr*

DESCRIPTION

ltpcli is a background ‘‘daemon’’ task that receives LTP data transmission blocks, extracts bundles from the received blocks, and passes them to the bundle protocol agent on the local ION node.

ltpcli is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol; the text of the command that is used to spawn the task must be provided at the time the ‘‘ltp’’ convergence layer protocol is added to the BP database. The convergence layer input task is terminated by **bpadmin** in response to an ’x’ (STOP) command. **ltpcli** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the LTP convergence layer protocol.

EXIT STATUS

0 **ltpcli** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **ltpcli**.

1 **ltpcli** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **ltpcli**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ltpcli can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such ltp duct.
: No LTP induct matching *local_node_nbr* has been added to the BP database. Use **bpadmin** to stop the LTP convergence-layer protocol, add the induct, and then restart the LTP protocol.

CLI task is already started for this duct.
: Redundant initiation of **ltpcli**.

ltpcli can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP operations.

ltpcli can’t open client access.
: Another task has already opened the client service for BP over LTP.

ltpcli can’t create receiver thread
: Operating system error. Check errtext, correct problem, and restart LTP.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *ltpadmin* (1), *ltprc* (5), *ltpclo* (1)

NAME

ltpclo – LTP–based BP convergence layer adapter output task

SYNOPSIS

ltpclo [–]*remote_node_nbr*

DESCRIPTION

ltpclo is a background ‘‘daemon’’ task that extracts bundles from the queues of segments ready for transmission via LTP to the remote bundle protocol agent identified by *remote_node_nbr* and passes them to the local LTP engine for aggregation, segmentation, and transmission to the remote node. If *remote_node_nbr* is preceded by a ’–’ character, then all LTP transmission performed by **ltpclo** will be ‘‘green’’ (unreliable) transmission, without acknowledgment and retransmission.

Note that **ltpclo** is not a ‘‘promiscuous’’ convergence layer daemon: it can transmit bundles only to the node for which it is configured, so scheme configuration directives that cite this outduct need only provide the protocol name and the outduct name (the remote node number) as specified on the command line when **ltpclo** is started.

ltpclo is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **ltpclo** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the LTP convergence layer protocol.

EXIT STATUS

0 **ltpclo** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart the BRSC protocol.

1 **ltpclo** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart the BRSC protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ltpclo can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such ltp duct.
: No LTP outduct with duct name matching *remote_node_nbr* has been added to the BP database. Use **bpadmin** to stop the LTP convergence-layer protocol, add the outduct, and then restart the LTP protocol.

CLO task is already started for this duct.
: Redundant initiation of **ltpclo**.

ltpclo can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP operations.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *ltpadmin* (1), *ltprc* (5), *ltpcli* (1)

NAME

sstcpcli – DTN simple TCP convergence layer input task

SYNOPSIS

stcpcli *local_hostname*[:*local_port_nbr*]

DESCRIPTION

stcpcli is a background ‘‘daemon’’ task comprising 1 + N threads: one that handles TCP connections from remote **stcpclo** tasks, spawning sockets for data reception from those tasks, plus one input thread for each spawned socket to handle data reception over that socket.

The connection thread simply accepts connections on a TCP socket bound to *local_hostname* and *local_port_nbr* and spawns reception threads. The default value for *local_port_nbr*, if omitted, is 4556.

Each reception thread receives bundles over the associated connected socket. Each bundle received on the connection is preceded by a 32–bit unsigned integer in network byte order indicating the length of the bundle. The received bundles are passed to the bundle protocol agent on the local ION node.

stcpcli is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol; the text of the command that is used to spawn the task must be provided at the time the ‘‘stcp’’ convergence layer protocol is added to the BP database. The convergence layer input task is terminated by **bpadmin** in response to an ’x’ (STOP) command. **stcpcli** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the STCP convergence layer protocol.

EXIT STATUS

0 **stcpcli** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **stcpcli**.

1 **stcpcli** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **stcpcli**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

stcpcli can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such stcp duct.
: No STCP induct matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the STCP convergence-layer protocol, add the induct, and then restart the STCP protocol.

CLI task is already started for this duct.
: Redundant initiation of **stcpcli**.

Can’t get IP address for host
: Operating system error. Check errtext, correct problem, and restart STCP.

Can’t open TCP socket
: Operating system error. Check errtext, correct problem, and restart STCP.

Can’t initialize socket
: Operating system error. Check errtext, correct problem, and restart STCP.

stcpcli can’t create access thread
: Operating system error. Check errtext, correct problem, and restart STCP.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *stcpclo* (1)

NAME

stcpclo – DTN simple TCP convergence layer adapter output task

SYNOPSIS

stcpclo *remote_hostname*[:*remote_port_nbr*]

DESCRIPTION

stcpclo is a background ‘‘daemon’’ task that connects to a remote node’s TCP socket at *remote_hostname* and *remote_port_nbr*. It then begins extracting bundles from the queues of bundles ready for transmission via TCP to this remote bundle protocol agent and transmitting those bundles over the connected socket to that node. Each transmitted bundle is preceded by a 32–bit integer in network byte order indicating the length of the bundle.

If not specified, *remote_port_nbr* defaults to 4556.

Note that **stcpclo** is not a ‘‘promiscuous’’ convergence layer daemon: it can transmit bundles only to the node to which it is connected, so scheme configuration directives that cite this outduct need only provide the protocol name and the outduct name as specified on the command line when **stcpclo** is started.

stcpclo is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **stcpclo** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the STCP convergence layer protocol.

EXIT STATUS

0 **stcpclo** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart the STCP protocol.

1 **stcpclo** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart the STCP protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

stcpclo can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such stcp duct.
: No STCP outduct with duct name matching *remote_hostname* and *remote_port_nbr* has been added to the BP database. Use **bpadmin** to stop the STCP convergence-layer protocol, add the outduct, and then restart the STCP protocol.

CLO task is already started for this duct.
: Redundant initiation of **stcpclo**.

Can’t get IP address for host
: Operating system error. Check errtext, correct problem, and restart STCP.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *stcpcli* (1)

NAME

tcpcli – DTN TCPCL–compliant convergence layer input task

SYNOPSIS

tcpcli *local_hostname*[:*local_port_nbr*]

DESCRIPTION

tcpcli is a background ‘‘daemon’’ task comprising 1 + N threads: one that handles TCP connections from remote **tcpclo** tasks, spawning sockets for data reception from those tasks, plus one input thread for each spawned socket to handle data reception over that socket.

The connection thread simply accepts connections on a TCP socket bound to *local_hostname* and *local_port_nbr* and spawns reception threads. The default value for *local_port_nbr*, if omitted, is 4556.

Each time a connection is established, the end-points will first exchange contact headers, because connection parameters need to be negotiated. **tcpcli** records the acknowledgement flags, reactive fragmentation flag and negative acknowledgements flag in the contact header it receives from its peer **tcpclo** task.

Each reception thread receives bundles over the associated connected socket. Each bundle received on the connection is preceded by message type, fragmentation flags, and size represented as an SDNV. The received bundles are passed to the bundle protocol agent on the local ION node.

tcpcli is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol; the text of the command that is used to spawn the task must be provided at the time the ‘‘tcp’’ convergence layer protocol is added to the BP database. The convergence layer input task is terminated by **bpadmin** in response to an ’x’ (STOP) command. **tcpcli** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the TCP convergence layer protocol.

EXIT STATUS

0 **tcpcli** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **tcpcli**.

1 **tcpcli** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **tcpcli**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

tcpcli can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such tcp duct.
: No TCP induct matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the TCP convergence-layer protocol, add the induct, and then restart the TCP protocol.

CLI task is already started for this duct.
: Redundant initiation of **tcpcli**.

Can’t get IP address for host
: Operating system error. Check errtext, correct problem, and restart TCP.

Can’t open TCP socket
: Operating system error. Check errtext, correct problem, and restart TCP.

Can't initialize socket
: Operating system error. Check errtext, correct problem, and restart TCP.

tcpcli can't create access thread
: Operating system error. Check errtext, correct problem, and restart TCP.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *tcpclo* (1)

NAME

tcpclo – DTN TCPCL–compliant convergence layer adapter output task

SYNOPSIS

tcpclo *remote_hostname*[:*remote_port_nbr*]

DESCRIPTION

tcpclo is a background ''daemon'' task that connects to a remote node's TCP socket at *remote_hostname* and *remote_port_nbr*. It sends a contact header, and it records the acknowledgement flag, reactive fragmentation flag and negative acknowledgements flag in the contact header it receives from its peer **tcpcli** task. It then begins extracting bundles from the queues of bundles ready for transmission via TCP to this remote bundle protocol agent and transmitting those bundles over the connected socket to that node. Each transmitted bundle is preceded by message type, segmentation flags, and an SDNV indicating the size of the bundle (in bytes).

If not specified, *remote_port_nbr* defaults to 4556.

Note that **tcpclo** is not a ''promiscuous'' convergence layer daemon: it can transmit bundles only to the node to which it is connected, so scheme configuration directives that cite this outduct need only provide the protocol name and the outduct name as specified on the command line when **tcpclo** is started.

tcpclo is spawned automatically by **bpadmin** in response to the 's' (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an 'x' (STOP) command. **tcpclo** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the TCP convergence layer protocol.

EXIT STATUS

0 **tcpclo** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart the TCPCL protocol.

1 **tcpclo** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart the TCPCL protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

tcpclo can't attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such tcp duct.
: No TCP outduct with duct name matching *remote_hostname* and *remote_port_nbr* has been added to the BP database. Use **bpadmin** to stop the TCP convergence-layer protocol, add the outduct, and then restart the TCP protocol.

CLO task is already started for this duct.
: Redundant initiation of **tcpclo**.

Can't get IP address for host
: Operating system error. Check errtext, correct problem, and restart TCP.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *tcpcli* (1)

NAME

udpcli – UDP–based BP convergence layer input task

SYNOPSIS

udpcli *local_hostname*[:*local_port_nbr*]

DESCRIPTION

udpcli is a background ‘‘daemon’’ task that receives UDP datagrams via a UDP socket bound to *local_hostname* and *local_port_nbr*, extracts bundles from those datagrams, and passes them to the bundle protocol agent on the local ION node.

If not specified, port number defaults to 4556.

The convergence layer input task is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol; the text of the command that is used to spawn the task must be provided at the time the ‘‘udp’’ convergence layer protocol is added to the BP database. The convergence layer input task is terminated by **bpadmin** in response to an ’x’ (STOP) command. **udpcli** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the UDP convergence layer protocol.

EXIT STATUS

0 **udpcli** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **udpcli**.

1 **udpcli** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **udpcli**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

udpcli can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No such udp duct.
: No UDP induct matching *local_hostname* and *local_port_nbr* has been added to the BP database. Use **bpadmin** to stop the UDP convergence-layer protocol, add the induct, and then restart the UDP protocol.

CLI task is already started for this duct.
: Redundant initiation of **udpcli**.

Can’t get IP address for host
: Operating system error. Check errtext, correct problem, and restart UDP.

Can’t open UDP socket
: Operating system error. Check errtext, correct problem, and restart UDP.

Can’t initialize socket
: Operating system error. Check errtext, correct problem, and restart UDP.

udpcli can’t create receiver thread
: Operating system error. Check errtext, correct problem, and restart UDP.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *udpclo* (1)

NAME

udpclo – UDP–based BP convergence layer output task

SYNOPSIS

udpclo

DESCRIPTION

udpclo is a background ‘‘daemon’’ task that extracts bundles from the queues of bundles ready for transmission via UDP to remote bundle protocol agents, encapsulates them in UDP datagrams, and sends those datagrams to the appropriate remote UDP sockets as indicated by the host names and UDP port numbers (destination induct names) associated with the bundles by the routing daemons that enqueued them.

Note that **udpclo** is a ‘‘promiscuous’’ CLO daemon, able to transmit bundles to any UDP destination induct. Its duct name is ’*’ rather than the induct name of any single UDP destination induct to which it might be dedicated, so scheme configuration directives that cite this outduct must provide destination induct IDs. For the UDP convergence-layer protocol, destination induct IDs are identical to induct names, i.e., they are of the form *local_hostname*[:*local_port_nbr*].

udpclo is spawned automatically by **bpadmin** in response to the ’s’ (START) command that starts operation of the Bundle Protocol, and it is terminated by **bpadmin** in response to an ’x’ (STOP) command. **udpclo** can also be spawned and terminated in response to START and STOP commands that pertain specifically to the UDP convergence layer protocol.

EXIT STATUS

0 **udpclo** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **bpadmin** to restart **udpclo**.

1 **udpclo** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **bpadmin** to restart **udpclo**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

udpclo can’t attach to BP.
: **bpadmin** has not yet initialized Bundle Protocol operations.

No memory for UDP buffer in udpclo.
: ION system error. Check errtext, correct problem, and restart UDP.

No such udp duct.
: No UDP outduct with duct name ’*’ has been added to the BP database. Use **bpadmin** to stop the UDP convergence-layer protocol, add the outduct, and then restart the UDP protocol.

CLO task is already started for this engine.
: Redundant initiation of **udpclo**.

CLO can’t open UDP socket
: Operating system error. Check errtext, correct problem, and restart **udpclo**.

CLO *write()* error on socket
: Operating system error. Check errtext, correct problem, and restart **udpclo**.

Bundle is too big for UDP CLA.
: Configuration error: bundles that are too large for UDP transmission (i.e., larger than 65535 bytes) are being enqueued for **udpclo**. Change routing.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpadmin (1), *bprc* (5), *udpcli* (1)

NAME

bssStreamingApp – Bundle Streaming Service transmission test program

SYNOPSIS

bssStreamingApp *own_endpoint_ID destination_endpoint_ID* [*class_of_service*]

DESCRIPTION

bssStreamingApp uses BSS to send streaming data over BP from *own_endpoint_ID* to bssrecv listening at *destination_endpoint_ID*. *class_of_service* is as specified for *bptrace* (1); if omitted, bundles are sent at BP's standard priority (1).

The bundles issued by **bssStreamingApp** all have 65000–byte payloads, where the ASCII representation of a positive integer (increasing monotonically from 0, by 1, throughout the operation of the program) appears at the start of each payload. All bundles are sent with custody transfer requested, with time-to-live set to 1 day. The application meters output by sleeping for 12800 microseconds after issuing each bundle.

Use CTRL-C to terminate the program.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bssrecv (1), *bss* (3)

NAME

bssrecv – Bundle Streaming Service reception test program

SYNOPSIS

bssrecv

DESCRIPTION

bssrecv uses BSS to acquire streaming data from **bssStreamingApp**.

bssrecv is a menu-driven interactive test program, run from the operating system shell prompt. The program enables the user to begin and end a session of BSS data acquisition from **bssStreamingApp**, displaying the data as it arrives in real time; to replay data acquired during the current session; and to replay data acquired during a prior session.

The user must provide values for three parameters in order to initiate the acquisition or replay of data from **bssStreamingApp**:

BSS database name
: All data acquired by the BSS session thread will be written to a BSS ''database'' comprising three files: table, list, and data. The name of the database is the root name that is common to the three files, e.g., *db3*.tbl, *db3*.lst, *db3*.dat would be the three files making up the *db3* BSS database.

path name
: All three files of the selected BSS database must reside in the same directory of the file system; the path name of that directory is required.

endpoint ID
: In order to acquire streaming data issued by **bssStreamingApp**, the **bssrecv** session thread must open the BP endpoint to which that data is directed. For this purpose, the ID of that endpoint is needed.

bssrecv offers the following menu options:

1. Open BSS Receiver in playback mode
   bssrecv prompts the user for the three parameter values noted above, then opens the indicated BSS database for replay of the data in that database.

2. Start BSS receiving thread
   bssrecv prompts the user for the three parameter values noted above, then starts a background session thread to acquire data into the indicated database. Each bundle that is acquired is passed to a display function that prints a single line consisting of N consecutive '*' characters, where N is computed as the data number at the start of the bundle's payload data, modulo 150. Note that the database is **not** open for replay at this time.

3. Run BSS receiver thread
   bssrecv prompts the user for the three parameter values noted above, then starts a background session thread to acquire data into the indicated database (displaying the data as described for option 2 above) and also opens the database for replay.

4. Close current playback session
   bssrecv closes the indicated BSS database, terminating replay access.

5. Stop BSS receiving thread
   bssrecv terminates the current background session thread. Replay access to the BSS database, if currently open, is **not** terminated.

6. Stop BSS Receiver
   bssrecv terminates the current background session thread. Replay access to the BSS database, if currently open, is also terminated.

7. Replay session
   bssrecv prompts the user for the start and end times bounding the reception interval that is to be replayed, then displays all data within that interval in both forward and reverse time order. The display function performed for this purpose is the same one that is exercised during real-time

acquisition of streaming data.

8. Exit
   bssrecv terminates.

EXIT STATUS

0 **bssrecv** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bssStreamingApp (1), *bss* (3)

NAME

bpcp – A remote copy utility for delay tolerant networks utilizing NASA JPL's Interplanetary Overlay Network (ION)

SYNOPSIS

bpcp [–dqr | –v] [–L *bundle_lifetime*] [–C *custody_on/off*] [–S *class_of_service*] [*host1*:]*file1* ... [*host2*:]*file2*

DESCRIPTION

bpcp copies files between hosts utilizing NASA JPL's Interplanetary Overlay Network (ION) to provide a delay tolerant network. File copies from local to remote, remote to local, or remote to remote are permitted. **bpcp** depends on ION to do any authentication or encryption of file transfers. All covergence layers over which **bpcp** runs MUST be reliable.

The options are permitted as follows:

–d Debug output. Repeat for increased verbosity.
–q Quiet. Do not output status messages.
–r Recursive.
–v Display version information.

–L *bundle_lifetime*
Bundle lifetime in seconds. Default is 86400 seconds (1 day).

–C *BP_custody*
Acceptable values are ON/OFF,YES/NO,1/0. Default is ON.

–S *class_of_service*
Bundle Protocol Class of Service for this transfer. Available options are:

0 Bulk Priority
1 Standard Priority
2 Expedited Priority

Default is Standard Priority.

bpcp utilizes CFDP to preform the actual file transfers. This has several important implications. First, ION's CFDP implementation requires that reliable convergence layers be used to transfer the data. Second, file permissions are not transferred. Files will be made executable on copy. Third, symbolic links are ignored for local to remote transfers and their target is copied for remote transfers. Fourth, all hosts must be specified using ION's IPN naming scheme.

In order to preform remote to local transfers or remote to remote transfers, **bpcpd** must be running on the remote hosts. However, **bpcp** should NOT be run simultaneously with **bpcpd** or **cfdptest**.

EXIT STATUS

0 **bpcp** terminated normally.

1 **bpcp** terminated abnormally. Check console and the **ion.log** file for error messages.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpcpd (1), *ion* (3), *cfdptest* (1)

NAME

bpcpd – ION Delay Tolerant Networking remote file copy daemon

SYNOPSIS

bpcpd [–d | –v]

DESCRIPTION

bpcpd is the daemon for **bpcp**. Together these programs copy files between hosts utilizing NASA JPL's Interplanetary Overlay Network (ION) to provide a delay tolerant network.

The options are permitted as follows:

–d Debug output. Repeat for increased verbosity.

–v Display version information.

bpcpd must be running in order to copy files from this host to another host (i.e. remote to local). Copies in the other direction (local to remote) do not require **bpcpd**. Further, **bpcpd** should NOT be run simultaneously with **bpcp** or **cfdptest**.

EXIT STATUS

0 **bpcpd** terminated normally.

1 **bpcpd** terminated abnormally. Check console and the **ion.log** file for error messages.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

bpcp (1), *ion* (3), *cfdptest* (1)

NAME

bputa – BP–based CFDP UT–layer adapter

SYNOPSIS

bputa

DESCRIPTION

bputa is a background ‘‘daemon’’ task that sends and receives CFDP PDUs encapsulated in DTN bundles.

The task is spawned automatically by **cfdpadmin** in response to the ’s’ command that starts operation of the CFDP protocol; the text of the command that is used to spawn the task must be provided as a parameter to the ’s’ command. The link service input task is terminated by **cfdpadmin** in response to an ’x’ (STOP) command.

EXIT STATUS

0 **bputa** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **cfdpadmin** to restart **bputa**.

1 **bputa** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **cfdpadmin** to restart **bputa**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

CFDP can’t attach to BP.
bpadmin has not yet initialized BP protocol operations.

CFDP can’t open own endpoint.
Most likely another bputa task is already running. Use **cfdpadmin** to stop CFDP and restart.

CFDP can’t get Bundle Protocol SAP.
Most likely a BP configuration problem. Use **bpadmin** to stop BP and restart.

bputa can’t attach to CFDP.
cfdpadmin has not yet initialized CFDP protocol operations.

bputa can’t dequeue outbound CFDP PDU; terminating.
Possible system error. Check ion.log for additional diagnostic messages.

bputa can’t send PDU in bundle; terminating.
Possible system error. Check ion.log for additional diagnostic messages.

bputa can’t track PDU; terminating.
Possible system error. Check ion.log for additional diagnostic messages.

bputa bundle reception failed.
Possible system error; reception thread terminates. Check ion.log for additional diagnostic messages.

bputa can’t receive bundle ADU.
Possible system error; reception thread terminates. Check ion.log for additional diagnostic messages.

bputa can’t handle bundle delivery.
Possible system error; reception thread terminates. Check ion.log for additional diagnostic messages.

bputa can’t handle inbound PDU.
Possible system error; reception thread terminates. Check ion.log for additional diagnostic messages.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

cfdpadmin (1), *bpadmin* (1)

NAME

cfdpadmin – ION's CCSDS File Delivery Protocol (CFDP) administration interface

SYNOPSIS

cfdpadmin [*commands_filename* | .]

DESCRIPTION

cfdpadmin configures, starts, manages, and stops CFDP operations for the local ION node.

It operates in response to CFDP configuration commands found in the file *commands_filename*, if provided; if not, **cfdpadmin** prints a simple prompt (:) so that the user may type commands directly into standard input. If *commands_filename* is a period (.), the effect is the same as if a command file containing the single command 'x' were passed to **cfdpadmin** — that is, the ION node's *cfdpclock* task and UT layer service task (nominally *bputa*) are stopped.

The format of commands for *commands_filename* can be queried from **cfdpadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *cfdprc* (5).

EXIT STATUS

0 Successful completion of CFDP administration.

EXAMPLES

cfdpadmin

Enter interactive CFDP configuration command entry mode.

cfdpadmin host1.cfdprc

Execute all configuration commands in *host1.cfdprc*, then terminate immediately.

cfdpadmin .

Stop all CFDP operations on the local node.

FILES

See *cfdprc* (5) for details of the CFDP configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the cfdprc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **cfdpadmin**. Otherwise **cfdpadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

cfdpadmin can't attach to ION.

There is no SDR data store for *cfdpadmin* to use. You should run *ionadmin* (1) first, to set up an SDR data store for ION.

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **cfdpadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *cfdprc* (5) for details.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

cfdprc (5)

NAME

cfdpclock – CFDP daemon task for managing scheduled events

SYNOPSIS

cfdpclock

DESCRIPTION

cfdpclock is a background ‘‘daemon’’ task that periodically performs scheduled CFDP activities. It is spawned automatically by **cfdpadmin** in response to the ’s’ command that starts operation of the CFDP protocol, and it is terminated by **cfdpadmin** in response to an ’x’ (STOP) command.

Once per second, **cfdpclock** takes the following action:

First it scans all inbound file delivery units (FDUs). For each one whose check timeout deadline has passed, it increments the check timeout count and resets the check timeout deadline. For each one whose check timeout count exceeds the limit configured for this node, it invokes the Check Limit Reached fault handling procedure.

Then it scans all outbound FDUs. For each one that has been Canceled, it cancels all extant PDU bundles and sets transmission progress to the size of the file, simulating the completion of transmission. It destroys each outbound FDU whose transmission is completed.

EXIT STATUS

0 **cfdpclock** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **cfdpadmin** to restart **cfdpclock**.

1 **cfdpclock** was unable to attach to CFDP protocol operations, probably because **cfdpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

cfdpclock can’t initialize CFDP.
cfdpadmin has not yet initialized CFDP protocol operations.

Can’t dispatch events.
An unrecoverable database error was encountered. **cfdpclock** terminates.

Can’t manage links.
An unrecoverable database error was encountered. **cfdpclock** terminates.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

cfdpadmin (1)

NAME

cfdptest – CFDP test shell for ION

SYNOPSIS

cfdptest [*commands_filename*]

DESCRIPTION

cfdptest provides a mechanism for testing CFDP file transmission. It can be used in either scripted or interactive mode. All bundles containing CFDP PDUs are sent with custody transfer requested and with all bundle status reporting disabled.

When scripted with *commands_filename*, **cfdptest** operates in response to CFDP management commands contained in the provided commands file. Each line of text in the file is interpreted as a single command comprising several tokens: a one-character command code and, in most cases, one or more command arguments of one or more characters. The commands configure and initiate CFDP file transmission operations.

If no file is specified, **cfdptest** instead offers the user an interactive ''shell'' for command entry. **cfdptest** prints a prompt string (": ") to stdout, accepts strings of text from stdin, and interprets each string as a command.

The supported **cfdptest** commands (whether interactive or scripted) are as follows:

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

h An alternate form of the **help** command.

d <destination CFDP entity ID number>
The **destination** command. This command establishes the CFDP entity to which the next file transmission operation will be directed. CFDP entity numbers in ION are, by convention, the same as BP node numbers.

f <source file path name>
The **from** command. This command identifies the file that will be transmitted when the next file transmission operation is commanded.

t <destination file path name>
The **to** command. This command provides the name for the file that will be created at the receiving entity when the next file transmission operation is commanded.

l <lifetime in seconds>
The **time-to-live** command. This command establishes the time-to-live for all subsequently issued bundles containing CFDP PDUs. If not specified, the default value 86400 (1 day) is used.

p <priority>
The **priority** command. This command establishes the priority (class of service) for all subsequently issued bundles containing CFDP PDUs. Valid values are 0, 1, and 2. If not specified, priority is 1.

o <ordinal>
The **ordinal** command. This command establishes the ''ordinal'' (sub-priority within priority 2) for all subsequently issued bundles containing CFDP PDUs. Valid values are 0–254. If not specified, ordinal is 0.

m <mode>
The **mode** command. This command establishes the transmission mode (''best-effort'' or assured) for all subsequently issued bundles containing CFDP PDUs. Valid values are 0 (assured, reliable, with reliability provided by a reliable DTN convergence layer protocol), 1 (best-effort, unreliable), and 2 (assured, reliable, but with reliability provided by BP custody transfer). If not specified, transmission mode is 0.

g <srrflags>
The **srrflags** command. This command establishes the BP status reporting that will be requested for all subsequently issued bundles containing CFDP PDUs. *srrflags* must be a status reporting flags string as defined for *bptrace* (1): a sequence of status report flags, separated by commas, with no embedded whitespace. Each status report flag must be one of the following: rcv, ct, fwd, dlv, del.

c <criticality>
The **criticality** command. This command establishes the criticality for all subsequently issued bundles containing CFDP PDUs. Valid values are 0 (not critical) and 1 (critical). If not specified, criticality is 0.

r <action code nbr> <first path name> <second path name>
The **filestore request** command. This command adds a filestore request to the metadata that will be issued when the next file transmission operation is commanded. Action code numbers are:

0 = create file
1 = delete file
2 = rename file
3 = append file
4 = replace file
5 = create directory
6 = remove directory
7 = deny file
8 = deny directory

u '<message text>'
The **user message** command. This command adds a user message to the metadata that will be issued when the next file transmission operation is commanded.

& The **send** command. This command initiates file transmission as configured by the most recent preceding **d**, **f**, and **t** commands.

^ The **cancel** command. This command cancels the most recently initiated file transmission.

% The **suspend** command. This command suspends the most recently initiated file transmission.

$ The **resume** command. This command resumes the most recently initiated file transmission.

The **report** command. This command reports on the most recently initiated file transmission.

q The **quit** command. Terminates the cfdptest program.

cfdptest in interactive mode also spawns a CFDP event handling thread. The event thread receives CFDP service indications and simply prints lines of text to stdout to announce them.

NOTE that when **cfdptest** runs in scripted mode it does **not** spawn an event handling thread, which makes it possible for the CFDP events queue to grow indefinitely unless some other task consumes and reports on the events. One simple solution is to run an interactive **cfdptest** task in background, simply to keep the event queue cleared, while scripted non-interactive **cfdptest** tasks are run in the foreground.

EXIT STATUS

0 **cfdptest** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

FILES

See above for details on valid *commands_filename* commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **cfdptest** are written to the ION log file *ion.log*.

Can't open command file...
: The file identified by *commands_filename* doesn't exist.

cfdptest can't initialize CFDP.
: **cfdpadmin** has not yet initialized CFDP operations.

Can't put FDU.
: The attempt to initiate file transmission failed. See the ION log for additional diagnostic messages from the CFDP library.

Failed getting CFDP event.
: The attempt to retrieve a CFDP service indication failed. See the ION log for additional diagnostic messages from the CFDP library.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

cfdpadmin (1), *cfdp* (3)

NAME

dgr2file – DGR reception test program

SYNOPSIS

dgr2file

DESCRIPTION

dgr2file uses DGR to receive multiple copies of the text of a file transmitted by **file2dgr**, writing each copy of the file to the current working directory. The name of each file written by **dgr2file** is file_copy_*cycleNbr*, where *cycleNbr* is initially zero and is increased by 1 every time **dgr2file** closes the file it is currently writing and opens a new one.

Upon receiving a DGR datagram from **file2dgr**, **dgr2file** extracts the content of the datagram (either a line of text from the file that is being transmitted by **file2dgr** or else an EOF string indicating the end of that file). It appends each extracted line of text to the local copy of that file that **dgr2file** is currently writing. When the extracted datagram content is an EOF string (the ASCII text "*** End of the file ***"), **dgr2file** closes the file it is writing, increments *cycleNbr*, opens a new copy of the file for writing, and prints the message "working on cycle *cycleNbr*."

dgr2file always receives datagrams at port 2101.

EXIT STATUS

0 **dgr2file** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

can't open dgr service
: Operating system error. Check errtext, correct problem, and rerun.

can't open output file
: Operating system error. Check errtext, correct problem, and rerun.

dgr_receive failed
: Operating system error. Check errtext, correct problem, and rerun.

can't write to output file
: Operating system error. Check errtext, correct problem, and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

file2dgr (1), *dgr* (3)

NAME

file2dgr – DGR transmission test program

SYNOPSIS

file2dgr *remoteHostName fileName* [*nbrOfCycles*]

DESCRIPTION

file2dgr uses DGR to send *nbrOfCycles* copies of the text of the file named *fileName* to the **dgr2file** process running on the computer identified by *remoteHostName*. If not specified (or if less than 1), *nbrOfCycles* defaults to 1. After sending all lines of the file, **file2dgr** sends a datagram containing an EOF string (the ASCII text "*** End of the file ***") before reopening the file and starting transmission of the next copy.

When all copies of the file have been sent, **file2dgr** prints a performance report:

```
Bytes sent = I<byteCount>, usec elapsed = I<elapsedTime>.

Sending I<dataRate> bits per second.
```

EXIT STATUS

0 **file2dgr** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **file2dgr** are written to the ION log file *ion.log*.

Can't open dgr service.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't open input file
: Operating system error. Check errtext, correct problem, and rerun.

Can't acquire DGR working memory.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't reopen input file
: Operating system error. Check errtext, correct problem, and rerun.

Can't read from input file
: Operating system error. Check errtext, correct problem, and rerun.

dgr_send failed.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

file2dgr (1), *dgr* (3)

NAME

file2sdr – SDR data ingestion test program

SYNOPSIS

file2sdr *configFlags fileName*

DESCRIPTION

file2sdr stress-tests SDR data ingestion by repeatedly writing all text lines of the file named *fileName* to one of a series of non-volatile linked lists created in a test SDR data store named "testsdr*configFlags*". By incorporating the data store configuration into the name (e.g., ''testsdr14") we make it relatively easy to perform comparative testing on SDR data stores that are identical aside from their configuration settings.

The operation of **file2sdr** is cyclical: a new linked list is created each time the program finishes copying the file's text lines and starts over again. If you use ^C to terminate **file2sdr** and then restart it, the program resumes operation at the point where it left off.

After writing each line to the current linked list, **file2sdr** gives a semaphore to indicate that the list is now non-empty. This is mainly for the benefit of the complementary test program *sdr2file* (1).

At the end of each cycle **file2sdr** appends a final EOF line to the current linked list, containing the text "*** End of the file ***", and prints a brief performance report:

```
Processing I<lineCount> lines per second.
```

EXIT STATUS

0 **file2sdr** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **file2sdr** are written to the ION log file *ion.log*.

Can't use sdr.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't create semaphore.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

SDR transaction failed.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can't open input file
: Operating system error. Check errtext, correct problem, and rerun.

Can't reopen input file
: Operating system error. Check errtext, correct problem, and rerun.

Can't read from input file
: Operating system error. Check errtext, correct problem, and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

sdr2file (1), *sdr* (3)

NAME

file2sm – shared–memory linked list data ingestion test program

SYNOPSIS

file2sm *fileName*

DESCRIPTION

file2sm stress-tests shared-memory linked list data ingestion by repeatedly writing all text lines of the file named *fileName* to a shared-memory linked list that is the root object of a PSM partition named ‘‘file2sm’’.

After writing each line to the linked list, **file2sm** gives a semaphore to indicate that the list is now non-empty. This is mainly for the benefit of the complementary test program *sm2file* (1).

The operation of **file2sm** is cyclical. After copying all text lines of the source file to the linked list, **file2sm** appends an EOF line to the linked list, containing the text ‘‘*** End of the file ***’’, and prints a brief performance report:

```
Processing I<lineCount> lines per second.
```

Then it reopens the source file and starts appending the file’s text lines to the linked list again.

EXIT STATUS

0 **file2sm** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Can’t attach to shared memory
: Operating system error. Check errtext, correct problem, and rerun.

Can’t manage shared memory.
: PSM error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can’t create shared memory list.
: smlist error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can’t create semaphore.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

Can’t open input file
: Operating system error. Check errtext, correct problem, and rerun.

Can’t reopen input file
: Operating system error. Check errtext, correct problem, and rerun.

Can’t read from input file
: Operating system error. Check errtext, correct problem, and rerun.

Ran out of memory.
: Nominal behavior. **sm2file** is not extracting data from the linked list quickly enough to prevent it from growing to consume all memory allocated to the test partition.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

sm2file (1), *smlist* (3), *psm* (3)

NAME

ionadmin – ION node administration interface

SYNOPSIS

ionadmin [*commands_filename* | .]

DESCRIPTION

ionadmin configures, starts, manages, and stops the ION node on the local computer.

It configures the node and sets (and reports on) global operational settings for the DTN protocol stack on the local computer in response to ION configuration commands found in *commands_filename*, if provided; if not, **ionadmin** prints a simple prompt (:) so that the user may type commands directly into standard input. If *commands_filename* is a period (.), the effect is the same as if a command file containing the single command 'x' were passed to **ionadmin** — that is, the ION node's *rfxclock* task is stopped.

The format of commands for *commands_filename* can be queried from **ionadmin** by entering the command 'h' or '?' at the prompt. The commands are documented in *ionrc* (5).

Note that *ionadmin* always computes a congestion forecast immediately before exiting. The result of this forecast — maximum projected occupancy of the DTN protocol traffic allocation in ION's SDR database — is retained for application flow control purposes: if maximum projected occupancy is the entire protocol traffic allocation, then a message to this effect is logged and no new bundle origination by any application will be accepted until a subsequent forecast that predicts no congestion is computed. (Congestion forecasts are constrained by *horizon* times, which can be established by commands issued to *ionadmin*. One way to re-enable data origination temporarily while long-term traffic imbalances are being addressed is to declare a congestion forecast horizon in the near future, before congestion would occur if no adjustments were made.)

EXIT STATUS

0 Successful completion of ION node administration.

EXAMPLES

ionadmin

Enter interactive ION configuration command entry mode.

ionadmin host1.ion

Execute all configuration commands in *host1.ion*, then terminate immediately.

FILES

Status and diagnostic messages from **ionadmin** and from other software that utilizes the ION node are nominally written to a log file in the current working directory within which **ionadmin** was run. The log file is typically named **ion.log**.

See also *ionconfig* (5) and *ionrc* (5).

ENVIRONMENT

Environment variables ION_NODE_LIST_DIR and ION_NODE_WDNAME can be used to enable the operation of multiple ION nodes on a single workstation computer. See section 2.1.3 of the ION Design and Operations Guide for details.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the ionrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **ionadmin**. Otherwise **ionadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the log file:

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

ionadmin SDR definition failed.

A node initialization command was executed, but an SDR database already exists for the indicated node. It is likely that an ION node is already running on this computer or that destruction of a previously started the previous ION node was incomplete. For most ION installations, incomplete node destruction can be repaired by (a) killing all ION processes that are still running and then (b) using **ipcrm** to remove all SVr4 IPC objects owned by ION.

ionadmin can't get SDR parms.

A node initialization command was executed, but the *ion_config_filename* passed to that command contains improperly formatted commands. Please see *ionconfig* (5) for further details.

Various errors that don't cause **ionadmin** to fail but are noted in the log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename*. Please see *ionrc* (5) for details.

BUGS

If the *ion_config_filename* parameter passed to a node initialization command refers to a nonexistent filename, then **ionadmin** uses default values are used rather than reporting an error in the command line argument.

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ionrc (5), *ionconfig* (5)

NAME

ionsecadmin – ION security policy administration interface

SYNOPSIS

ionsecadmin [*commands_filename*]

DESCRIPTION

ionsecadmin configures and manages the ION security policy database on the local computer.

It configures and manages the ION security policy database on the local computer in response to ION configuration commands found in *commands_filename*, if provided; if not, **ionsecadmin** prints a simple prompt (:) so that the user may type commands directly into standard input.

The format of commands for *commands_filename* can be queried from **ionsecadmin** by entering the command 'h' or '?' at the prompt. The commands are documented in *ionsecrc* (5).

EXIT STATUS

0 Successful completion of ION security policy administration.

EXAMPLES

ionsecadmin

Enter interactive ION security policy administration command entry mode.

ionsecadmin host1.ionsecrc

Execute all configuration commands in *host1.ionsecrc*, then terminate immediately.

FILES

Status and diagnostic messages from **ionsecadmin** and from other software that utilizes the ION node are nominally written to a log file in the current working directory within which **ionsecadmin** was run. The log file is typically named **ion.log**.

See also *ionsecrc* (5).

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the ionrc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **ionsecadmin**. Otherwise **ionsecadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the log file:

Can't open command file...

The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **ionsecadmin** to fail but are noted in the log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename*. Please see *ionsecrc* (5) for details.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ionsecrc (5)

NAME

owltsim – one−way light time transmission delay simulator

SYNOPSIS

owltsim *config_filename* [−v]

DESCRIPTION

owltsim delays delivery of data between pairs of ION nodes by specified lengths of time, simulating the signal propagation delay imposed by distance between the nodes.

Its operation is configured by delay simulation configuration lines in the file identified by *config_filename*. A pair of threads is created for each line in the file: one that receives UDP datagrams on a specified port and queues them in a linked list, and a second that later removes queued datagrams from the linked list and sends them on to a specified UDP port on a specified network host.

Each configuration line must be of the following form:

to from my_port# dest_host dest_port# owlt modulus

to identifies the receiving node.

This parameter is purely informational, intended to make **owltsim**'s printed messages more helpful to the user.

from identifies the sending node.

A value of '*' may be used to indicate ''all nodes''. Again, this parameter is purely informational, intended to make **owltsim**'s printed messages more helpful to the user.

my_port# identifies **owltsim**'s receiving port for this traffic.

dest_host is a hostname identifying the computer to which **owltsim** will transmit this traffic.

dest_port# identifies the port to which **owltsim** will transmit this traffic.

owlt specifies the number of seconds to wait before forwarding each received datagram.

modulus controls the artificial random data loss imposed on this traffic by **owltsim**.

A value of '0' specifies ''no random data loss''. Any other modulus value N causes **owltsim** to randomly drop (i.e., not transmit upon expiration of the delay interval) one out of every N packets.

The optional **−v** (''verbose'') parameter causes **owltsim** to print a message whenever it receives, sends, or drops (due to artificial random data loss) a datagram.

Note that error conditions may cause one delay simulation (a pair of threads) to terminate without terminating any others.

owltsim is designed to run indefinitely. To terminate the program, just use control-C to kill it.

EXIT STATUS

0 Nominal termination.
1 Termination due to an error condition, as noted in printed messages.

EXAMPLES

Here is a sample owltsim configuration file:

2 7 5502 ptl07.jpl.nasa.gov 5001 75 0
7 2 5507 ptl02.jpl.nasa.gov 5001 75 16

This file indicates that **owltsim** will receive on port 5502 the ION traffic from node 2 that is destined for node 7, which will receive it at port 5001 on the computer named ptl07.jpl.nasa.gov; 75 seconds of delay (simulating a distance of 75 light seconds) will be imposed on this transmission activity, and **owltsim** will not simulate any random data loss.

In the reverse direction, **owltsim** will receive on port 5507 the ION traffic from node 7 that is destined for node 2, which will receive it at port 5001 on the computer named ptl02.jpl.nasa.gov; 75 seconds of delay

will again be imposed on this transmission activity, and **owltsim** will randomly discard (i.e., not transmit upon expiration of the transmission delay interval) one datagram out of every 16 received at this port.

FILES

Not applicable.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be printed to stdout:

owltsim can't open configuration file
: The program terminates.

owltsim failed on fscanf
: Failure on reading the configuration file. The program terminates.

owltsim stopped malformed config file line *line_number*.
: Failure on parsing the configuration file. The program terminates.

owltsim can't spawn receiver thread
: The program terminates.

owltsim out of memory.
: The program terminates.

owltsim can't open reception socket
: The program terminates.

owltsim can't initialize reception socket
: The program terminates.

owltsim can't open transmission socket
: The program terminates.

owltsim can't initialize transmission socket
: The program terminates.

owltsim can't spawn timer thread
: The program terminates.

owltsim can't acquire datagram
: Datagram transmission failed. This causes the threads for the affected delay simulation to terminate, without terminating any other threads.

owltsim failed on send
: Datagram transmission failed. This causes the threads for the affected delay simulation to terminate, without terminating any other threads.

at *time* owltsim LOST a dg of length *length* from *sending node* destined for *receiving node* due to ECONNREFUSED.
: This is an informational message. Due to an apparent bug in Internet protocol implementation, transmission of a datagram on a connected UDP socket occasionally fails. **owltsim** does not attempt to retransmit the affected datagram.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

udplsi (1), *udplso* (1)

POD ERRORS

Hey! **The above document had some coding errors, which are explained below:**

Around line 24:
You can’t have =items (as at line 29) unless the first thing after the =over is an =item

NAME

owlttb – one–way light time transmission delay simulator

SYNOPSIS

owlttb *own_uplink_port# own_downlink_port# dest_uplink_IP_address dest_uplink_port# dest_downlink_IP_address dest_downlink_port# owlt_sec*. [–v]

DESCRIPTION

owlttb delays delivery of data between an NTTI and a NetAcquire box (or two, one for uplink and one for downlink) by a specified length of time, simulating the signal propagation delay imposed by distance between the nodes.

Its operation is configured by the command-line parameters, except that the delay interval itself may be changed while the program is running. **owlttb** offers a command prompt (:), and when a new value of one-way light time is entered at this prompt the new delay interval takes effect immediately.

own_uplink_port# identifies the port on **owlttb** accepts the NTTI's TCP connection for uplink traffic (i.e., data destined for the NetAcquire box).
own_downlink_port# identifies the port on **owlttb** accepts the NTTI's TCP connection for downlink traffic (i.e., data issued by the NetAcquire box).
dest_uplink_IP_address is the IP address (a dotted string) identifying the NetAcquire box to which **owlttb** will transmit uplink traffic.
dest_uplink_port# identifies the TCP port to which **owlttb** will connect in order to transmit uplink traffic to NetAcquire.
dest_downlink_IP_address is the IP address (a dotted string) identifying the NetAcquire box from which **owlttb** will receive downlink traffic.
dest_downlink_port# identifies the TCP port to which **owlttb** will connect in order to receive downlink traffic from NetAcquire.
owlt specifies the number of seconds to wait before forwarding each received segment of TCP traffic.

The optional **–v** (''verbose'') parameter causes **owlttb** to print a message whenever it receives, sends, or discards (due to absence of a connected downlink client) a segment of TCP traffic.

owlttb is designed to run indefinitely. To terminate the program, just use control-C to kill it or enter ''q'' at the prompt.

EXIT STATUS

0 Nominal termination.
1 Termination due to an error condition, as noted in printed messages.

EXAMPLES

Here is a sample owlttb command:

owlttb 2901 2902 137.7.8.19 10001 137.7.8.19 10002 75

This command indicates that **owlttb** will accept an uplink traffic connection on port 2901, forwarding the received uplink traffic to port 10001 on the NetAcquire box at 137.7.8.19, and it will accept a downlink traffic connection on port 2902, delivering over that connection all downlink traffic that it receives from connecting to port 10002 on the NetAcquire box at 137.7.8.19. 75 seconds of delay (simulating a distance of 75 light seconds) will be imposed on this transmission activity.

FILES

Not applicable.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be printed to stdout:

owlttb can't spawn uplink thread
The program terminates.

owlttb can't spawn uplink sender thread
The program terminates.

owlttb can't spawn downlink thread
The program terminates.

owlttb can't spawn downlink receiver thread
The program terminates.

owlttb can't spawn downlink sender thread
The program terminates.

owlttb fgets failed
The program terminates.

owlttb out of memory.
The program terminates.

owlttb lost uplink client.
This is an informational message. The NTTI may reconnect at any time.

owlttb lost downlink client
This is an informational message. The NTTI may reconnect at any time.

owlttb can't open TCP socket to NetAcquire
The program terminates.

owlttb can't connect TCP socket to NetAcquire
The program terminates.

owlttb *write()* error on socket
The program terminates if it was writing to NetAcquire; otherwise it simply recognizes that the client NTTI has disconnected.

owlttb *read()* error on socket
The program terminates.

owlttb can't open uplink dialup socket
The program terminates.

owlttb can't initialize uplink dialup socket
The program terminates.

owlttb can't open downlink dialup socket
The program terminates.

owlttb can't initialize downlink dialup socket
The program terminates.

owlttb *accept()* failed
The program terminates.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

NAME

psmshell – PSM memory management test shell

SYNOPSIS

psmshell *partition_size*

DESCRIPTION

psmshell allocates a region of *partition_size* bytes of system memory, places it under PSM management, and offers the user an interactive ‘‘shell’’ for testing various PSM management functions.

psmshell prints a prompt string (‘‘: ’’) to stdout, accepts a command from stdin, executes the command (possibly printing a diagnostic message), then prints another prompt string and so on.

The locations of objects allocated from the PSM-managed region of memory are referred to as ‘‘cells’’ in psmshell operations. That is, when an object is to be allocated, a cell number in the range 0–99 must be specified as the notional ‘‘handle’’ for that object, for use in future commands.

The following commands are supported:

h The **help** command. Causes **psmshell** to print a summary of available commands. Same effect as the **?** command.

? Another **help** command. Causes **psmshell** to print a summary of available commands. Same effect as the **h** command.

m *cell_nbr size*
The **malloc** command. Allocates a large-pool object of the indicated size and associates that object with *cell_nbr*.

z *cell_nbr size*
The **zalloc** command. Allocates a small-pool object of the indicated size and associates that object with *cell_nbr*.

p *cell_nbr*
The **print** command. Prints the address (i.e., the offset within the managed block of memory) of the object associated with *cell_nbr*.

f *cell_nbr*
The **free** command. Frees the object associated with *cell_nbr*, returning the space formerly occupied by that object to the appropriate free block list.

u The **usage** command. Prints a partition usage report, as per *psm_report* (3).

q The **quit** command. Frees the allocated system memory in the managed block and terminates **psmshell**.

EXIT STATUS

0 **psmshell** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

IPC initialization failed.
ION system error. Investigate, correct problem, and try again.

psmshell: can’t allocate space; quitting.
Insufficient available system memory for selected partition size.

psmshell: can’t allocate test variables; quitting.
Insufficient available system memory for selected partition size.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

psm (3)

NAME

psmwatch – PSM memory partition activity monitor

SYNOPSIS

psmwatch *shared_memory_key memory_size partition_name interval count* [verbose]

DESCRIPTION

For *count* interations, **psmwatch** sleeps *interval* seconds and then invokes the *psm_print_trace()* function (see *psm* (3)) to report on PSM dynamic memory management activity in the PSM-managed shared memory partition identified by *shared_memory_key* during that interval. If the optional **verbose** parameter is specified, the printed PSM activity trace will be verbose as described in *psm* (3).

To prevent confusion, the specified *memory_size* and *partition_name* are compared to those declared when this shared memory partition was initially managed; if they don't match, **psmwatch** immediately terminates.

If *interval* is zero, **psmwatch** merely prints a current usage summary for the indicated shared-memory partition and terminates.

psmwatch is helpful for detecting and diagnosing memory leaks. For debugging the ION protocol stack:

shared_memory_key
: Normally ''65281'', but might be overridden by the value of wmKey in the .ionconfig file used to configure the node under study.

memory_size
: As given by the value of wmKey in the .ionconfig file used to configure the node under study. If this value is not stated in the .ionconfig file, the default value is ''5000000''.

partition_name
: Always ''ionwm''.

EXIT STATUS

0 **psmwatch** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't attach to psm.
: ION system error. One possible cause is that ION has not yet been initialized on the local computer; run *ionadmin* (1) to correct this.

Can't start trace.
: Insufficient ION working memory to contain trace information. Reinitialize ION with more memory.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

psm (3), *sdrwatch* (1)

NAME

rfxclock – ION daemon task for managing scheduled events

SYNOPSIS

rfxclock

DESCRIPTION

rfxclock is a background ‘‘daemon’’ task that periodically applies scheduled changes in node connectivity and range to the ION node’s database. It is spawned automatically by **ionadmin** in response to the ’s’ command that starts operation of the ION node infrastructure, and it is terminated by **ionadmin** in response to an ’x’ (STOP) command.

Once per second, **rfxclock** takes the following action:

For each neighboring node that has been refusing custody of bundles sent to it to be forwarded to some destination node, to which no such bundle has been sent for at least N seconds (where N is twice the one-way light time from the local node to this neighbor), **rfxclock** turns on a *probeIsDue* flag authorizing transmission of the next such bundle in hopes of learning that this neighbor is now able to accept custody.

Then **rfxclock** purges the database of all range and contact information that is no longer applicable, based on the stop times of the records.

Finally, **rfxclock** applies to the database all range and contact information that is currently applicable, i.e., those records whose start times are before the current time and whose stop times are in the future.

EXIT STATUS

0 **rfxclock** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ionadmin** to restart **rfxclock**.

1 **rfxclock** was unable to attach to the local ION node, probably because **ionadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

rfxclock can’t attach to ION.
: **ionadmin** has not yet initialized the ION database.

Can’t apply ranges.
: An unrecoverable database error was encountered. **rfxclock** terminates.

Can’t apply contacts.
: An unrecoverable database error was encountered. **rfxclock** terminates.

Can’t purge ranges.
: An unrecoverable database error was encountered. **rfxclock** terminates.

Can’t purge contacts.
: An unrecoverable database error was encountered. **rfxclock** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ionadmin (1)

NAME

sdr2file – SDR data extraction test program

SYNOPSIS

sdr2file *configFlags*

DESCRIPTION

sdr2file stress-tests SDR data extraction by retrieving and deleting all text file lines inserted into a test SDR data store named "testsdr*configFlags*" by the complementary test program *file2sdr* (1).

The operation of **sdr2file** echoes the cyclical operation of **file2sdr**: each linked list created by **file2sdr** is used to create in the current working directory a copy of **file2sdr**'s original source text file. The name of each file written by **sdr2file** is file_copy_*cycleNbr*, where *cycleNbr* identifies the linked list from which the file's text lines were obtained.

sdr2file may catch up with the data ingestion activity of **file2sdr**, in which case it blocks (taking the **file2sdr** test semaphore) until the linked list it is currently draining is no longer empty.

EXIT STATUS

0 **sdr2file** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Can't use sdr.
: ION system error. Check for diagnostics in the ION log file *ion.log*.

Can't create semaphore.
: ION system error. Check for diagnostics in the ION log file *ion.log*.

SDR transaction failed.
: ION system error. Check for diagnostics in the ION log file *ion.log*.

Can't open output file
: Operating system error. Check errtext, correct problem, and rerun.

can't write to output file
: Operating system error. Check errtext, correct problem, and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

file2sdr (1), *sdr* (3)

NAME

sdrmend – SDR corruption repair utility

SYNOPSIS

sdrmend *sdr_name config_flags heap_words heap_key path_name* [*restartCmd restartLatency*]

DESCRIPTION

The **sdrmend** program simply invokes the *sdr_reload_profile()* function (see *sdr* (3)) to effect necessary repairs in a potentially corrupt SDR, e.g., due to the demise of a program that had an SDR transaction in progress at the moment it crashed.

Note that **sdrmend** need not be run to repair ION's data store in the event of a hardware reboot: restarting ION will automatically reload the data store's profile. **sdrmend** is needed only when it is desired to repair the data store without requiring all ION software to terminate and restart.

EXIT STATUS

0 **sdrmend** has terminated successfully.

1 **sdrmend** has terminated unsuccessfully. See diagnostic messages in the **ion.log** log file for details.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can't initialize the SDR system.

Probable operations error: ION appears not to be initialized, in which case there is no point in running **sdrmend**.

Can't reload profile for SDR.

ION system error. See earlier diagnostic messages posted to **ion.log** for details. In this event it is unlikely that **sdrmend** can be run successfully, and it is also unlikely that it would have any effect if it did run successfully.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

sdr (3), *ionadmin* (1)

NAME

sdrwatch – SDR non−volatile data store activity monitor

SYNOPSIS

sdrwatch *sdr_name interval count* [verbose]

DESCRIPTION

For *count* interations, **sdrwatch** sleeps *interval* seconds and then invokes the *sdr_print_trace()* function (see *sdr* (3)) to report on SDR data storage management activity in the SDR data store identified by *sdr_name* during that interval. If the optional **verbose** parameter is specified, the printed SDR activity trace will be verbose as described in *sdr* (3).

If *interval* is zero, **sdrwatch** merely prints a current usage summary for the indicated data store and terminates.

sdrwatch is helpful for detecting and diagnosing storage space leaks. For debugging the ION protocol stack, *sdr_name* is normally ‘‘ion’’ but might be overridden by the value of sdrName in the .ionconfig file used to configure the node under study.

EXIT STATUS

0 **sdrwatch** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

Can’t attach to sdr.
: ION system error. One possible cause is that ION has not yet been initialized on the local computer; run *ionadmin* (1) to correct this.

Can’t start trace.
: Insufficient ION working memory to contain trace information. Reinitialize ION with more memory.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

sdr (3), *psmwatch* (1)

NAME

sm2file – shared–memory linked list data extraction test program

SYNOPSIS

sm2file

DESCRIPTION

sm2file stress-tests shared-memory linked list data extraction by retrieving and deleting all text file lines inserted into a shared-memory linked list that is the root object of a PSM partition named ‘‘file2sm’’.

The operation of **sm2file** echoes the cyclical operation of **file2sm**: the EOF lines inserted into the linked list by **file2sm** punctuate the writing of files that are copies of **file2sm**'s original source text file. The name of each file written by **sm2file** is file_copy_*cycleNbr*, where *cycleNbr* is, in effect, the count of EOF lines encountered in the linked list up to the point at which the writing of this file began.

sm2file may catch up with the data ingestion activity of **file2sm**, in which case it blocks (taking the **file2sm** test semaphore) until the linked list is no longer empty.

EXIT STATUS

0 **sm2file** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

can’t attach to shared memory
: Operating system error. Check errtext, correct problem, and rerun.

Can’t manage shared memory.
: PSM error. Check for earlier diagnostics describing the cause of the error; correct problem and rerun.

Can’t create shared memory list.
: PSM error. Check for earlier diagnostics describing the cause of the error; correct problem and rerun.

Can’t create semaphore.
: ION system error. Check for earlier diagnostics describing the cause of the error; correct problem and rerun.

Can’t open output file
: Operating system error. Check errtext, correct problem, and rerun.

can’t write to output file
: Operating system error. Check errtext, correct problem, and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

file2sm (1), *smlist* (3), *psm* (3)

NAME

smlistsh – shared–memory linked list test shell

SYNOPSIS

smlistsh *partition_size*

DESCRIPTION

smlistsh attaches to a region of system memory (allocating it if necessary, and placing it under PSM management as necessary) and offers the user an interactive ‘‘shell’’ for testing various shared-memory linked list management functions.

smlistsh prints a prompt string (‘‘: ’’) to stdout, accepts a command from stdin, executes the command (possibly printing a diagnostic message), then prints another prompt string and so on.

The following commands are supported:

h The **help** command. Causes **smlistsh** to print a summary of available commands. Same effect as the **?** command.

? Another **help** command. Causes **smlistsh** to print a summary of available commands. Same effect as the **h** command.

k The **key** command. Computes and prints an unused shared-memory key, for possible use in attaching to a shared-memory region.

+ *key_value size*
The **attach** command. Attaches **smlistsh** to a region of shared memory. *key_value* identifies an existing shared-memory region, in the event that you want to attach to an existing shared-memory region (possibly created by another **smlistsh** process running on the same computer). To create and attach to a new shared-memory region that other processes can attach to, use a *key_value* as returned by the **key** command and supply the *size* of the new region. If you want to create and attach to a new shared-memory region that is for strictly private use, use −1 as key and supply the *size* of the new region.

– The **detach** command. Detaches **smlistsh** from the region of shared memory it is currently using, but does not free any memory.

n The **new** command. Creates a new shared-memory list to operate on, within the currently attached shared-memory region. Prints the address of the list.

s *list_address*
The **share** command. Selects an existing shared-memory list to operate on, within the currently attached shared-memory region.

a *element_value*
The **append** command. Appends a new list element, containing *element_value*, to the list on which **smlistsh** is currently operating.

p *element_value*
The **prepend** command. Prepends a new list element, containing *element_value*, to the list on which **smlistsh** is currently operating.

w The **walk** command. Prints the addresses and contents of all elements of the list on which **smlistsh** is currently operating.

f *element_value*
The **find** command. Finds the list element that contains *element_value*, within the list on which **smlistsh** is currently operating, and prints the address of that list element.

d *element_address*
The **delete** command. Deletes the list element located at *element_address*.

r The **report** command. Prints a partition usage report, as per *psm_report*(3).

q The **quit** command. Detaches **smlistsh** from the region of shared memory it is currently using (without freeing any memory) and terminates **smlistsh**.

EXIT STATUS

0 **smlistsh** has terminated.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

No diagnostics apply.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

smlist (3)

NAME

dccplsi – DCCP–based LTP link service input task

SYNOPSIS

dccplsi {*local_hostname* | @}[:*local_port_nbr*]

DESCRIPTION

dccplsi is a background ‘‘daemon’’ task that receives DCCP datagrams via a DCCP socket bound to *local_hostname* and *local_port_nbr*, extracts LTP segments from those datagrams, and passes them to the local LTP engine. Host name ‘‘@’’ signifies that the host name returned by *hostname* (1) is to be used as the socket’s host name. If not specified, port number defaults to 1113.

The link service input task is spawned automatically by **ltpadmin** in response to the ’s’ command that starts operation of the LTP protocol; the text of the command that is used to spawn the task must be provided as a parameter to the ’s’ command. The link service input task is terminated by **ltpadmin** in response to an ’x’ (STOP) command.

EXIT STATUS

0 **dccplsi** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **dccplsi**.

1 **dccplsi** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **ltpadmin** to restart **dccplsi**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dccplsi can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

LSI task is already started.
: Redundant initiation of **dccplsi**.

LSI can’t open DCCP socket. This probably means DCCP is not supported on your system.
: Operating system error. This probably means that you are not using an operating system that supports DCCP. Make sure that you are using a current Linux kernel and that the DCCP modules are being compiled. Check errtext, correct problem, and restart **dccplsi**.

LSI can’t initialize socket.
: Operating system error. Check errtext, correct problem, and restart **dccplsi**.

LSI can’t create listener thread.
: Operating system error. Check errtext, correct problem, and restart **dccplsi**.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *dccplso* (1), *owltsim* (1)

NAME

dccplso – DCCP–based LTP link service output task

SYNOPSIS

dccplso {*remote_engine_hostname* | @}[:*remote_port_nbr*] *remote_engine_nbr*

DESCRIPTION

dccplso is a background ‘‘daemon’’ task that extracts LTP segments from the queue of segments bound for the indicated remote LTP engine, encapsulates them in DCCP datagrams, and sends those datagrams to the indicated DCCP port on the indicated host. If not specified, port number defaults to 1113.

Each ‘‘span’’ of LTP data interchange between the local LTP engine and a neighboring LTP engine requires its own link service output task, such as **dccplso**. All link service output tasks are spawned automatically by **ltpadmin** in response to the ’s’ command that starts operation of the LTP protocol, and they are all terminated by **ltpadmin** in response to an ’x’ (STOP) command.

EXIT STATUS

0 **dccplso** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **dccplso**.

1 **dccplso** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **ltpadmin** to restart **dccplso**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

dccplso can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

No such engine in database.
: *remote_engine_nbr* is invalid, or the applicable span has not yet been added to the LTP database by **ltpadmin**.

LSO task is already started for this engine.
: Redundant initiation of **dccplso**.

LSO can’t create idle thread.
: Operating system error. Check errtext, correct problem, and restart **dccplso**.

LSO can’t open DCCP socket. This probably means DCCP is not supported on your system.
: Operating system error. This probably means that you are not using an operating system that supports DCCP. Make sure that you are using a current Linux kernel and that the DCCP modules are being compiled. Check errtext, correct problem, and restart **dccplso**.

LSO can’t connect DCCP socket.
: Remote host’s **dccplsi** isn’t listening or has terminated. Restart **dccplsi** on the remote host and then restart **dccplso**.

Segment is too big for DCCP LSO.
: Configuration error: segments that are too large for DCCP transmission (i.e., larger than 65535 bytes) are being enqueued for **dccplso**. Use **ltpadmin** to change maximum segment size for this span.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpmeter* (1), *dccplsi* (1), *owltsim* (1)

NAME

ltpadmin – ION Licklider Transmission Protocol (LTP) administration interface

SYNOPSIS

ltpadmin [*commands_filename* | .]

DESCRIPTION

ltpadmin configures, starts, manages, and stops LTP operations for the local ION node.

It operates in response to LTP configuration commands found in the file *commands_filename*, if provided; if not, **ltpadmin** prints a simple prompt (:) so that the user may type commands directly into standard input. If *commands_filename* is a period (.), the effect is the same as if a command file containing the single command 'x' were passed to **ltpadmin** — that is, the ION node's *ltpclock* task, *ltpmeter* tasks, and link service adapter tasks are stopped.

The format of commands for *commands_filename* can be queried from **ltpadmin** with the 'h' or '?' commands at the prompt. The commands are documented in *ltprc* (5).

EXIT STATUS

0 Successful completion of LTP administration.

EXAMPLES

ltpadmin
: Enter interactive LTP configuration command entry mode.

ltpadmin host1.ltp
: Execute all configuration commands in *host1.ltp*, then terminate immediately.

ltpadmin .
: Stop all LTP operations on the local node.

FILES

See *ltprc* (5) for details of the LTP configuration commands.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Note: all ION administration utilities expect source file input to be lines of ASCII text that are NL-delimited. If you edit the ltprc file on a Windows machine, be sure to **use dos2unix to convert it to Unix text format** before presenting it to **ltpadmin**. Otherwise **ltpadmin** will detect syntax errors and will not function satisfactorily.

The following diagnostics may be issued to the logfile ion.log:

ltpadmin can't attach to ION.
: There is no SDR data store for *ltpadmin* to use. You should run *ionadmin* (1) first, to set up an SDR data store for ION.

Can't open command file...
: The *commands_filename* specified in the command line doesn't exist.

Various errors that don't cause **ltpadmin** to fail but are noted in the **ion.log** log file may be caused by improperly formatted commands given at the prompt or in the *commands_filename* file. Please see *ltprc* (5) for details.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpmeter (1), *ltprc* (5)

NAME

ltpclock – LTP daemon task for managing scheduled events

SYNOPSIS

ltpclock

DESCRIPTION

ltpclock is a background ‘‘daemon’’ task that periodically performs scheduled LTP activities. It is spawned automatically by **ltpadmin** in response to the ’s’ command that starts operation of the LTP protocol, and it is terminated by **ltpadmin** in response to an ’x’ (STOP) command.

Once per second, **ltpclock** takes the following action:

First it manages the current state of all links (‘‘spans’’). In particular, it checks the age of the currently buffered session block for each span and, if that age exceeds the span’s configured aggregation time limit, gives the ‘‘buffer full’’ semaphore for that span to initiate block segmentation and transmission by **ltpmeter**.

In so doing, it also infers link state changes (‘‘link cues’’) from data rate changes as noted in the RFX database by **rfxclock**:

If the rate of transmission to a neighbor was zero but is now non-zero, then transmission to that neighbor is unblocked. The applicable ‘‘buffer empty’’ semaphore is given if no outbound block is being constructed (enabling start of a new transmission session) and the ‘‘segments ready’’ semaphore is given if the outbound segment queue is non-empty (enabling transmission of segments by the link service output task).

If the rate of transmission to a neighbor was non-zero but is now zero, then transmission to that neighbor is blocked — i.e., the semaphores triggering transmission will no longer be given.

If the imputed rate of transmission from a neighbor was non-zero but is now zero, then all timers affecting segment retransmission to that neighbor are suspended. This has the effect of extending the interval of each affected timer by the length of time that the timers remain suspended.

If the imputed rate of transmission from a neighbor was zero but is now non-zero, then all timers affecting segment retransmission to that neighbor are resumed.

Then **ltpclock** retransmits all unacknowledged checkpoint segments, report segments, and cancellation segments whose computed timeout intervals have expired.

EXIT STATUS

0 **ltpclock** terminated, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **ltpclock**.

1 **ltpclock** was unable to attach to LTP protocol operations, probably because **ltpadmin** has not yet been run.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ltpclock can’t initialize LTP.
ltpadmin has not yet initialized LTP protocol operations.

Can’t dispatch events.
An unrecoverable database error was encountered. **ltpclock** terminates.

Can’t manage links.
An unrecoverable database error was encountered. **ltpclock** terminates.

BUGS

Report bugs to <ion−bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpmeter* (1), *rfxclock* (1)

NAME

ltpcounter – LTP reception test program

SYNOPSIS

ltpcounter *client_ID* [*max_nbr_of_bytes*]

DESCRIPTION

ltpcounter uses LTP to receive service data units flagged with client service number *client_ID* from a remote **ltpdriver** client service process. When the total number of bytes of client service data it has received exceeds *max_nbr_of_bytes*, it terminates and prints reception and cancellation statistics. If *max_nbr_of_bytes* is omitted, the default limit is 2 billion bytes.

While receiving data, **ltpcounter** prints a 'v' character every 5 seconds to indicate that it is still alive.

EXIT STATUS

0 **ltpcounter** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

1 **ltpcounter** was unable to start, because it could not attach to the LTP protocol on the local node or could not open access to client service *clientId*.

In the former case, run **ltpadmin** to start LTP and try again.

In the latter case, some other client service task has already opened access to client service *clientId*. If no such task is currently running (e.g., it crashed while holding the client service open), use **ltpadmin** to stop and restart the LTP protocol.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **ltpcounter** are written to the ION log file *ion.log*.

ltpcounter can't initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

ltpcounter can't open client access.
: Another task has opened access to service client *clientId* and has not yet relinquished it.

Can't get LTP notice.
: LTP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpdriver* (1), *ltp* (3)

NAME

ltpdriver – LTP transmission test program

SYNOPSIS

ltpdriver *remoteEngineNbr clientId nbrOfCycles greenLength* [*totalLength*]

DESCRIPTION

ltpdriver uses LTP to send *nbrOfCycles* service data units of length indicated by *totalLength*, of which the trailing *greenLength* bytes are sent unreliably, to the **ltpcounter** client service process for client service number *clientId* attached to the remote LTP engine identified by *remoteEngineNbr*. If omitted, *length* defaults to 60000. If *length* is 1, the sizes of the transmitted service data units will be randomly selected multiples of 1024 in the range 1024 to 62464.

Whenever the size of the transmitted service data unit is less than or equal to *greenLength*, the entire SDU is sent unreliably.

When all copies of the file have been sent, **ltpdriver** prints a performance report.

EXIT STATUS

0 **ltpdriver** has terminated. Any problems encountered during operation will be noted in the **ion.log** log file.

1 **ltpdriver** was unable to start, because it could not attach to the LTP protocol on the local node. Run **ltpadmin** to start LTP, then try again.

FILES

The service data units transmitted by **ltpdriver** are sequences of text obtained from a file in the current working directory named ''ltpdriverAduFile'', which **ltpdriver** creates automatically.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

Diagnostic messages produced by **ltpdriver** are written to the ION log file *ion.log*.

ltpdriver can't initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

Can't create ADU file
: Operating system error. Check errtext, correct problem, and rerun.

Error writing to ADU file
: Operating system error. Check errtext, correct problem, and rerun.

ltpdriver can't create file ref.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

ltpdriver can't create ZCO.
: ION system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

ltpdriver can't send message.
: LTP span to the remote engine has been stopped.

ltp_send failed.
: LTP system error. Check for earlier diagnostic messages describing the cause of the error; correct problem and rerun.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpcounter* (1), *ltp* (3)

NAME

ltpmeter – LTP daemon task for aggregating and segmenting transmission blocks

SYNOPSIS

ltpmeter *remote_engine_nbr*

DESCRIPTION

ltpmeter is a background ‘‘daemon’’ task that manages the presentation of LTP segments to link service output tasks. Each ‘‘span’’ of LTP data interchange between the local LTP engine and a neighboring LTP engine requires its own **ltpmeter** task. All **ltpmeter** tasks are spawned automatically by **ltpadmin** in response to the ’s’ command that starts operation of the LTP protocol, and they are all terminated by **ltpadmin** in response to an ’x’ (STOP) command.

ltpmeter waits until its span’s current transmission block (the data to be transmitted during the transmission session that is currently being constructed) is ready for transmission, then divides the data in the span’s block buffer into segments and enqueues the segments for transmission by the span’s link service output task (giving the segments semaphore to unblock the link service output task as necessary), then reinitializes the span’s block buffer and starts another session (giving the ‘‘buffer empty’’ semaphore to unblock the client service task — nominally **ltpclo**, the LTP convergence layer output task for Bundle Protocol — as necessary).

ltpmeter determines that the current transmission block is ready for transmission by waiting until either (a) the aggregate size of all service data units in the block’s buffer exceeds the aggregation size limit for this span or (b) the length of time that the first service data unit in the block’s buffer has been awaiting transmission exceeds the aggregation time limit for this span. The ‘‘buffer full’’ semaphore is given when ION (either the *ltp_send()* function or the **ltpclock** daemon) determines that one of these conditions is true; **ltpmeter** simply waits for this semaphore to be given.

The initiation of a new session may also be blocked: the total number of transmission sessions that the local LTP engine may have open at a single time is limited (this is LTP flow control), and while the engine is at this limit no new sessions can be started. Availability of a session from the session pool is signaled by the ‘‘session’’ semaphore, which is given whenever a session is completed or canceled.

EXIT STATUS

0 **ltpmeter** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **ltpmeter**.

1 **ltpmeter** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **ltpadmin** to restart **ltpmeter**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

ltpmeter can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

No such engine in database.
: *remote_engine_nbr* is invalid, or the applicable span has not yet been added to the LTP database by **ltpadmin**.

ltpmeter task is already started for this engine.
: Redundant initiation of **ltpmeter**.

ltpmeter can’t start new session.
: An unrecoverable database error was encountered. **ltpmeter** terminates.

Can't take bufClosedSemaphore.
An unrecoverable database error was encountered. **ltpmeter** terminates.

Can't create extents list.
An unrecoverable database error was encountered. **ltpmeter** terminates.

Can't post ExportSessionStart notice.
An unrecoverable database error was encountered. **ltpmeter** terminates.

Can't finish session.
An unrecoverable database error was encountered. **ltpmeter** terminates.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpclock* (1)

NAME

udplsi – UDP–based LTP link service input task

SYNOPSIS

udplsi {*local_hostname* | @}[:*local_port_nbr*]

DESCRIPTION

udplsi is a background ‘‘daemon’’ task that receives UDP datagrams via a UDP socket bound to *local_hostname* and *local_port_nbr*, extracts LTP segments from those datagrams, and passes them to the local LTP engine. Host name ‘‘@’’ signifies that the host name returned by *hostname* (1) is to be used as the socket’s host name. If not specified, port number defaults to 1113.

The link service input task is spawned automatically by **ltpadmin** in response to the ’s’ command that starts operation of the LTP protocol; the text of the command that is used to spawn the task must be provided as a parameter to the ’s’ command. The link service input task is terminated by **ltpadmin** in response to an ’x’ (STOP) command.

EXIT STATUS

0 **udplsi** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **udplsi**.

1 **udplsi** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **ltpadmin** to restart **udplsi**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

udplsi can’t initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

LSI task is already started.
: Redundant initiation of **udplsi**.

LSI can’t open UDP socket
: Operating system error. Check errtext, correct problem, and restart **udplsi**.

LSI can’t initialize socket
: Operating system error. Check errtext, correct problem, and restart **udplsi**.

LSI can’t create receiver thread
: Operating system error. Check errtext, correct problem, and restart **udplsi**.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *udplso* (1), *owltsim* (1)

NAME

udplso – UDP–based LTP link service output task

SYNOPSIS

udplso {*remote_engine_hostname* | @}[:*remote_port_nbr*] [*txbps*] *remote_engine_nbr*

DESCRIPTION

udplso is a background "daemon" task that extracts LTP segments from the queue of segments bound for the indicated remote LTP engine, encapsulates them in UDP datagrams, and sends those datagrams to the indicated UDP port on the indicated host. If not specified, port number defaults to 1113.

UDP congestion can be controlled by setting udplso's rate of UDP datagram transmission *txbps* (transmission rate in bits per second) to the value that is supported by the underlying network.

Each "span" of LTP data interchange between the local LTP engine and a neighboring LTP engine requires its own link service output task, such as **udplso**. All link service output tasks are spawned automatically by **ltpadmin** in response to the 's' command that starts operation of the LTP protocol, and they are all terminated by **ltpadmin** in response to an 'x' (STOP) command.

EXIT STATUS

0 **udplso** terminated normally, for reasons noted in the **ion.log** file. If this termination was not commanded, investigate and solve the problem identified in the log file and use **ltpadmin** to restart **udplso**.

1 **udplso** terminated abnormally, for reasons noted in the **ion.log** file. Investigate and solve the problem identified in the log file, then use **ltpadmin** to restart **udplso**.

FILES

No configuration files are needed.

ENVIRONMENT

No environment variables apply.

DIAGNOSTICS

The following diagnostics may be issued to the **ion.log** log file:

udplso can't initialize LTP.
: **ltpadmin** has not yet initialized LTP protocol operations.

No such engine in database.
: *remote_engine_nbr* is invalid, or the applicable span has not yet been added to the LTP database by **ltpadmin**.

LSO task is already started for this engine.
: Redundant initiation of **udplso**.

LSO can't open UDP socket
: Operating system error. Check errtext, correct problem, and restart **udplso**.

LSO can't connect UDP socket
: Operating system error. Check errtext, correct problem, and restart **udplso**.

Segment is too big for UDP LSO.
: Configuration error: segments that are too large for UDP transmission (i.e., larger than 65535 bytes) are being enqueued for **udplso**. Use **ltpadmin** to change maximum segment size for this span.

BUGS

Report bugs to <ion–bugs@korgano.eecs.ohiou.edu>

SEE ALSO

ltpadmin (1), *ltpmeter* (1), *udplsi* (1), *owltsim* (1)

NAME

bp – Bundle Protocol communications library

SYNOPSIS

```
#include "bp.h"

[see description for available functions]
```

DESCRIPTION

The bp library provides functions enabling application software to use Bundle Protocol to send and receive information over a delay-tolerant network. It conforms to the Bundle Protocol specification as documented in Internet RFC 5050.

int bp_attach( )

Attaches the application to BP functionality on the local computer. Returns 0 on success, –1 on any error.

Note that all ION libraries and applications draw memory dynamically, as needed, from a shared pool of ION working memory. The size of the pool is established when ION node functionality is initialized by *ionadmin* (1). This is a precondition for initializing BP functionality by running *bpadmin* (1), which in turn is required in order for *bp_attach()* to succeed.

Sdr bp_get_sdr( )

Returns handle for the SDR data store used for BP, to enable creation and interrogation of bundle payloads (application data units).

void bp_detach( )

Terminates all access to BP functionality on the local computer.

int bp_open(char *eid, BpSAP *ionsapPtr)

Opens the application's access to the BP endpoint identified by *eid*, so that the application can take delivery of bundles destined for the indicated endpoint and can send bundles whose source is the indicated endpoint. On success, places a value in **ionsapPtr* that can be supplied to future bp function invocations and returns 0. Returns –1 on any error.

int bp_send(BpSAP sap, char *destEid, char *reportToEid, int lifespan, int classOfService, BpCustodySwitch custodySwitch, unsigned char srrFlags, int ackRequested, BpExtendedCOS *extendedCOS, Object adu, Object *newBundle)

Sends a bundle to the endpoint identified by *destEid*, from the source endpoint as provided to the *bp_open()* call that returned *sap*. When *sap* is NULL, the transmitted bundle is anonymous, i.e., the source of the bundle is not identified. This is legal, but anonymous bundles cannot be uniquely identified; custody transfer and status reporting therefore cannot be requested for an anonymous bundle.

reportToEid identifies the endpoint to which any status reports pertaining to this bundle will be sent; if NULL, defaults to the source endpoint.

lifespan is the maximum number of seconds that the bundle can remain in-transit (undelivered) in the network prior to automatic deletion.

classOfService is simply priority for now: BP_BULK_PRIORITY, BP_STD_PRIORITY, or BP_EXPEDITED_PRIORITY. If class-of-service flags are defined in a future version of Bundle Protocol, those flags would be OR'd with priority.

custodySwitch indicates whether or not custody transfer is requested for this bundle and, if so, whether or not the source node itself is required to be the initial custodian. The valid values are SourceCustodyRequired, SourceCustodyOptional, NoCustodyRequired. Note that custody transfer is possible only for bundles that are uniquely identified, so it cannot be requested for bundles for which BP_MINIMUM_LATENCY is requested, since BP_MINIMUM_LATENCY may result in the production of multiple identical copies of the same bundle. Similarly, custody transfer should never be requested for a ''loopback'' bundle, i.e., one whose destination node is the same as the source node: the received

bundle will be identical to the source bundle, both residing in the same node, so no custody acceptance signal can be applied to the source bundle and the source bundle will remain in storage until its TTL expires.

srrFlags, if non-zero, is the logical OR of the status reporting behaviors requested for this bundle: BP_RECEIVED_RPT, BP_CUSTODY_RPT, BP_FORWARDED_RPT, BP_DELIVERED_RPT, BP_DELETED_RPT.

ackRequested is a Boolean parameter indicating whether or not the recipient application should be notified that the source application requests some sort of application-specific end-to-end acknowledgment upon receipt of the bundle.

extendedCOS, if not NULL, is used to populate the Extended Class Of Service block for this bundle. The block's *ordinal* value is used to provide fine-grained ordering within "expedited" traffic: ordinal values from 0 (the default) to 254 (used to designate the most urgent traffic) are valid, with 255 reserved for custody signals. The value of the block's *flags* is the logical OR of the applicable extended class-of-service flags:

BP_MINIMUM_LATENCY designates the bundle as "critical" for the purposes of Contact Graph Routing.

BP_BEST_EFFORT signifies that non-reliable convergence-layer protocols, as available, may be used to transmit the bundle. Notably, the bundle may be sent as "green" data rather than "red" data when issued via LTP.

BP_FLOW_LABEL_PRESENT signifies whether or not the value of *flowLabel* in *extendedCOS* must be encoded into the ECOS block when the bundle is transmitted.

adu is the "application data unit" that will be conveyed as the payload of the new bundle. *adu* must be a "zero-copy object" (ZCO). ZCOs are normally created by invoking *ionCreateZco()*, which will block so long as insufficient ZCO storage space is available for creation of the requested ZCO (admission control); if non-blocking behavior is preferred, ZCOs may instead be created by *zco_create()*, which fails immediately if insufficient ZCO storage space is available.

The function returns 1 on success, 0 on user error, –1 on any system error. If 0 is returned, then an invalid argument value was passed to *bp_send()*; a message to this effect will have been written to the log file. If 1 is returned, then either the destination of the bundle was "dtn:none" (the bit bucket) or the ADU has been accepted and queued for transmission in a bundle; in the latter case (and only in this case) the address of the newly created bundle within the ION database is placed in *newBundle*, in case the bundle needs to be canceled in the future.

int bp_track(Object bundle, Object trackingElt)

Adds *trackingElt* to the list of "tracking" references in *bundle*. *trackingElt* must be the address of an SDR list element — whose data is the address of this same bundle — within some list of bundles that is privately managed by the application. Upon destruction of the bundle this list element will automatically be deleted, thus removing the bundle from the application's privately managed list of bundles. This enables the application to keep track of bundles that it is operating on without risk of inadvertently de-referencing the address of a nonexistent bundle.

void bp_untrack(Object bundle, Object trackingElt)

Removes *trackingElt* from the list of "tracking" references in *bundle*, if it is in that list. Does not delete *trackingElt* itself.

int bp_suspend(Object bundle)

Suspends transmission of *bundle*. Has no effect if bundle is "critical" (i.e., has got extended class of service BP_MINIMUM_LATENCY flag set) or if the bundle is already suspended. Otherwise, reverses the enqueuing of the bundle to its selected transmission outduct and places it in the "limbo" queue until the suspension is lifted by calling bp_resume. Returns 0 on success, –1 on any error.

int bp_resume(Object bundle)
Terminates suspension of transmission of *bundle*. Has no effect if bundle is ''critical'' (i.e., has got extended class of service BP_MINIMUM_LATENCY flag set) or is not suspended. Otherwise, removes the bundle from the ''limbo'' queue and queues it for route re-computation and re-queuing. Returns 0 on success, –1 on any error.

int bp_cancel(Object bundle)
Cancels transmission of *bundle*. If the indicated bundle is currently queued for forwarding, transmission, or retransmission, it is removed from the relevant queue and destroyed exactly as if its Time To Live had expired. Returns 0 on success, –1 on any error.

int bp_receive(BpSAP sap, BpDelivery *dlvBuffer, int timeoutSeconds)
Receives a bundle, or reports on some failure of bundle reception activity.

The ''result'' field of the dlvBuffer structure will be used to indicate the outcome of the data reception activity.

If at least one bundle destined for the endpoint for which this SAP is opened has not yet been delivered to the SAP, then the payload of the oldest such bundle will be returned in *dlvBuffer–>adu* and *dlvBuffer–>result* will be set to BpPayloadPresent. If there is no such bundle, *bp_receive()* blocks for up to *timeoutSeconds* while waiting for one to arrive.

If *timeoutSeconds* is BP_POLL (i.e., zero) and no bundle is awaiting delivery, or if *timeoutSeconds* is greater than zero but no bundle arrives before *timeoutSeconds* have elapsed, then *dlvBuffer–>result* will be set to BpReceptionTimedOut. If *timeoutSeconds* is BP_BLOCKING (i.e., –1) then *bp_receive()* blocks until either a bundle arrives or the function is interrupted by an invocation of *bp_interrupt()*.

dlvBuffer–>result will be set to BpReceptionInterrupted in the event that the calling process received and handled some signal other than SIGALRM while waiting for a bundle.

The application data unit delivered in the data delivery structure, if any, will be a ''zero-copy object'' reference. Use zco reception functions (see *zco* (3)) to read the content of the application data unit.

Be sure to call *bp_release_delivery()* after every successful invocation of *bp_receive()*.

The function returns 0 on success, –1 on any error.

void bp_interrupt(BpSAP sap)
Interrupts a *bp_receive()* invocation that is currently blocked. This function is designed to be called from a signal handler; for this purpose, *sap* may need to be obtained from a static variable.

void bp_release_delivery(BpDelivery *dlvBuffer, int releaseAdu)
Releases resources allocated to the indicated delivery. *releaseAdu* is a Boolean parameter: if non-zero, the ADU ZCO reference in *dlvBuffer* (if any) is destroyed, causing the ZCO itself to be destroyed if no other references to it remain.

void bp_close(BpSAP sap)
Terminates the application's access to the BP endpoint identified by the *eid* cited by the indicated service access point. The application relinquishes its ability to take delivery of bundles destined for the indicated endpoint and to send bundles whose source is the indicated endpoint.

SEE ALSO

bpadmin (1), *lgsend* (1), *lgagent* (1), *bpextensions* (3), *bprc* (5), *lgfile* (5)

NAME

bpextensions – interface for adding extensions to Bundle Protocol

SYNOPSIS

```
#include "bpextensions.c"
```

DESCRIPTION

ION's interface for extending the Bundle Protocol enables the definition of external functions that insert *extension* blocks into outbound bundles (either before or after the payload block), parse and record extension blocks in inbound bundles, and modify extension blocks at key points in bundle processing. All extension-block handling is statically linked into ION at build time, but the addition of an extension never requires that any standard ION source code be modified.

Standard structures for recording extension blocks — both in transient storage [memory] during bundle acquisition (AcqExtBlock) and in persistent storage [the ION database] during subsequent bundle processing (ExtensionBlock) — are defined in the *bpP.h* header file. In each case, the extension block structure comprises a block *type* code, block processing *flags*, possibly a list of *EID references*, an array of *bytes* (the serialized form of the block, for transmission), the *length* of that array, optionally an extension-specific opaque *object* whose structure is designed to characterize the block in a manner that's convenient for the extension processing functions, and the *size* of that object.

The definition of each extension is asserted in an ExtensionDef structure, also as defined in the *bpP.h* header file. Each ExtensionDef must supply:

The name of the extension. (Used in some diagnostic messages.)

The extension's block type code.

An indication as to whether the local node is to insert this extension block before (0) or after (1) the payload block when new bundles are locally sourced.

A pointer to an **offer** function.

A pointer to a **release** function.

A pointer to an **accept** function.

A pointer to a **check** function.

A pointer to a **record** function.

A pointer to a **clear** function.

A pointer to a **copy** function.

A pointer to a function to be called when **forwarding** a bundle containing this sort of block.

A pointer to a function to be called when **taking custody** of a bundle containing this sort of block.

A pointer to a function to be called when **enqueuing** for transmission a bundle containing this sort of block.

A pointer to a function to be called when a convergence-layer adapter **dequeues** a bundle containing this sort of block, before serializing it.

A pointer to a function to be called immediately before a convergence-layer adapter **transmits** a bundle containing this sort of block, after the bundle has been serialized.

All extension definitions must be coded into an array of ExtensionDef structures named *extensions*. The order of appearance of extension definitions in the extensions array determines the order in which extension blocks will be inserted into locally sourced bundles.

The standard extensions array — which is empty — is in the *noextensions.c* prototype source file. The procedure for extending the Bundle Protocol in ION is as follows:

1. Specify –DBP_EXTENDED in the Makefile's compiler command line when building the libbpP.c

library module.

2. Create a copy of the prototype extensions file, named ‘‘bpextensions.c’’, in a directory that is made visible to the Makefile’s libbpP.c compilation command line (by a –I parameter).

3. In the ‘‘external function declarations’’ area of ‘‘bpextensions.c’’, add ‘‘extern’’ function declarations identifying the functions that will implement your extension (or extensions).

4. Add one or more ExtensionDef structure initialization lines to the extensions array, referencing those declared functions.

5. Develop the implementations of those functions in one or more new source code files.

6. Add the object file or files for the new extension implementation source file (or files) to the Makefile’s command line for linking libbpP.so.

The function pointers supplied in each ExtensionDef must conform to the following specifications. NOTE that any function that modifies the *bytes* member of an ExtensionBlock or AckExtBlock **must** set the corresponding *length* to the new length of the *bytes* array, if changed.

int (*BpExtBlkOfferFn)(ExtensionBlock *blk, Bundle *bundle)
: Populates all fields of the indicated ExtensionBlock structure for inclusion in the indicated outbound bundle. This function is automatically called when a new bundle is locally sourced or upon acquisition of a remotely sourced bundle that does not contain an extension block of this type. The values of the extension block are typically expected to be a function of the state of the bundle, but this is extension-specific. If it is not appropriate to offer an extension block of this type as part of this bundle, then the *size*, *length*, *object*, and *bytes* members of *blk* must all be set to zero. If it is appropriate to offer such a block but no internal object representing the state of the block is needed, the *object* and *size* members of *blk* must be set to zero. The *type*, *blkProcFlags*, and *dataLength* members of *blk* must be populated by the implementation of the ‘‘offer’’ function, but the *length* and *bytes* members are typically populated by calling the BP library function *serializeExtBlk()*, which must be passed the block to be serialized (with *type*, *blkProcFlags* and *dataLength* already set), a Lyst of EID references (two list elements — offsets — per EID reference, if applicable; otherwise NULL), and a pointer to the extension-specific block data. The block’s *bytes* array and *object* (if present) must occupy space allocated from the ION database heap. Return zero on success, –1 on any system failure.

void (*BpExtBlkReleaseFn)(ExtensionBlock *blk)
: Releases all ION database space occupied by the *object* member of *blk*. This function is automatically called when a bundle is destroyed. Note that incorrect implementation of this function may result in a database space leak.

int (*BpExtBlkRecordFn)(ExtensionBlock *blk, AcqExtBlock *acqblk)
: Copies the *object* member of *acqblk* to ION database heap space and places the address of that non-volatile object in the *object* member of *blk*; also sets *size* in *blk*. This function is automatically called when an acquired bundle is accepted for forwarding and/or delivery. Return zero on success, –1 on any system failure.

int (*BpExtBlkCopyFn)(ExtensionBlock *newblk, ExtensionBlock *oldblk)
: Copies the *object* member of *oldblk* to ION database heap space and places the address of that new non-volatile object in the *object* member of *newblk*, also sets *size* in *newblk*. This function is automatically called when two copies of a bundle are needed, e.g., in the event that it must both be delivered to a local client and also fowarded to another node. Return zero on success, –1 on any system failure.

int (*BpExtBlkProcessFn)(ExtensionBlock *blk, Bundle *bundle, void *context)
: Performs some extension-specific transformation of the data encapsulated in *blk* based on the state of *bundle*. The transformation to be performed will typically vary depending on whether the identified function is the one that is automatically invoked upon forwarding the bundle, upon taking custody of the bundle, upon enqueuing the bundle for transmission, upon removing the bundle from the transmission queue, or upon transmitting the serialized bundle. The *context* argument may supply useful supplemental information; in particular, the context provided to the ON_DEQUEUE function will

comprise the name of the protocol for the duct from which the bundle has been dequeued, together with the EID of the neighboring node endpoint to which the bundle will be directly transmitted when serialized. The block-specific data in *blk* is located within *bytes* immediately after the header of the extension block; the length of the block's header is the difference between *length* and *dataLength*. Whenever the block's *blkProcFlags*, EID extensions, and/or block-specific data are altered, the *serializeExtBlk()* function should be called again to recalculate the size of the extension block and rebuild the *bytes* array. Return zero on success, –1 on any system failure.

int (*BpAcqExtBlkAcquireFn)(AcqExtBlock *acqblk, AcqWorkArea *work)
Populates the indicated AcqExtBlock structure with *size* and *object* for retention as part of the indicated inbound bundle. (The *type*, *blkProcFlags*, EID references (if any), *dataLength*, *length*, and *bytes* values of the structure are pre-populated with data as extracted from the serialized bundle.) This function is automatically called when an extension block of this type is encountered in the course of parsing and acquiring a bundle for local delivery and/or forwarding. If no internal object representing the state of the block is needed, the *object* member of *acqblk* must be set to NULL and the *size* member must be set to zero. If an *object* is needed for this block, it must occupy space that is allocated from ION working memory using **MTAKE** and its *size* must be indicated in *blk*. Return zero if the block is malformed (this will cause the bundle to be discarded), 1 if the block is successfully parsed, –1 on any system failure.

int (*BpAcqExtBlkCheckFn)(AcqExtBlock *acqblk, AcqWorkArea *work)
Examines the bundle in *work* to determine whether or not it is authentic, in the context of the indicated extension block. Return 1 if the block is determined to be inauthentic (this will cause the bundle to be discarded), zero if no inauthenticity is detected, –1 on any system failure.

void (*BpAcqExtBlkClearFn)(AcqExtBlock *acqblk)
Uses **MRELEASE** to release all ION working memory occupied by the *object* member of *acqblk*. This function is automatically called when acquisition of a bundle is completed, whether or not the bundle is accepted. Note that incorrect implementation of this function may result in a working memory leak.

UTILITY FUNCTIONS FOR EXTENSION PROCESSING

void discardExtensionBlock(AcqExtBlock *blk)
Deletes this block from the bundle acquisition work area prior to the recording of the bundle in the ION database.

void scratchExtensionBlock(ExtensionBlock *blk)
Deletes this block from the bundle after the bundle has been recorded in the ION database.

Object findExtensionBlock(Bundle *bundle, unsigned int type, unsigned int listIdx)
On success, returns the address of the ExtensionBlock in *bundle* for the indicated *type* and *listIdx*. If no such extension block exists, returns zero.

int serializeExtBlk(ExtensionBlock *blk, Lyst eidReferences, char *blockData)
Constructs an RFC5050–conformant serialized representation of this extension block in blk–>bytes. Returns 0 on success, –1 on an unrecoverable system error.

void suppressExtensionBlock(ExtensionBlock *blk)
Causes *blk* to be omitted when the bundle to which it is attached is serialized for transmission. This suppression remains in effect until it is reversed by *restoreExtensionBlock()*;

void restoreExtensionBlock(ExtensionBlock *blk)
Reverses the effect of *suppressExtensionBlock()*, enabling the block to be included when the bundle to which it is attached is serialized.

SEE ALSO

bp (3)

NAME

bss – Bundle Streaming Service library

SYNOPSIS

```
#include "bss.h"

typedef int (*RTBHandler)(time_t time, unsigned long count, char *buffer, int bu

[see description for available functions]
```

DESCRIPTION

The BSS library supports the streaming of data over delay-tolerant networking (DTN) bundles. The intent of the library is to enable applications that pass streaming data received in transmission time order (i.e., without time regressions) to an application-specific ‘‘display’’ function — notionally for immediate real-time display — but to store **all** received data (including out-of-order data) in a private database for playback under user control. The reception and real-time display of in-order data is performed by a background thread, leaving the application’s main (foreground) thread free to respond to user commands controlling playback or other application-specific functions.

The application-specific ‘‘display’’ function invoked by the background thread must conform to the RTBHandler type definition. It must return 0 on success, −1 on any error that should terminate the background thread. Only on return from this function will the background thread proceed to acquire the next BSS payload.

All data acquired by the BSS background thread is written to a BSS database comprising three files: table, list, and data. The name of the database is the root name that is common to the three files, e.g., *db3*.tbl, *db3*.lst, *db3*.dat would be the three files making up the *db3* BSS database. All three files of the selected BSS database must reside in the same directory of the file system.

Several replay navigation functions in the BSS library require that the application provide a navigation state structure of type bssNav as defined in the bss.h header file. The application is not reponsible for populating this structure; it’s strictly for the private use of the BSS library.

int bssOpen(char *bssName, char *path, char *eid)

Opens access to a BSS database, to enable data playback. *bssName* identifies the specific BSS database that is to be opened. *path* identifies the directory in which the database resides. *eid* is ignored. On any failure, returns −1. On success, returns zero.

int bssStart(char *bssName, char *path, char *eid, char *buffer, int bufLen, RTBHandler handler)

Starts a BSS data acquisition background thread. *bssName* identifies the BSS database into which data will be acquired. *path* identifies the directory in which that database resides. *eid* is used to open the BP endpoint at which the delivered BSS bundle payload contents will be acquired. *buffer* identifies a data acquisition buffer, which must be provided by the application, and *bufLen* indicates the length of that buffer; received bundle payloads in excess of this length will be discarded.

handler identifies the display function to which each in-order bundle payload will be passed. The *time* and *count* parameters passed to this function identify the received bundle, indicating the bundle’s creation timestamp time (in seconds) and counter value. The *buffer* and *bufLength* parameters indicate the location into which the bundle’s payload was acquired and the length of the acquired payload. *handler* must return −1 on any unrecoverable system error, 0 otherwise. A return value of −1 from *handler* will terminate the BSS data acquisition background thread.

On any failure, returns −1. On success, returns zero.

int bssRun(char *bssName, char *path, char *eid, char *buffer, int bufLen, RTBHandler handler)

A convenience function that performs both *bssOpen()* and *bssStart()*. On any failure, returns −1. On success, returns zero.

void *bssClose()*

Terminates data playback access to the most recently opened BSS database.

void *bssStop()*
Terminates the most recently initiated BSS data acquisition background thread.

void *bssExit()*
A convenience function that performs both *bssClose()* and *bssStop()*.

long bssRead(bssNav nav, char *data, int dataLen)
Copies the data at the current playback position in the database, as indicated by *nav*, into *data*; if the length of the data is in excess of *dataLen* then an error condition is asserted (i.e., –1 is returned). Note that *bssRead()* cannot be successfully called until *nav* has been populated, nominally by a preceding call to *bssSeek()*, *bssNext()*, or *bssPrev()*. Returns the length of data read, or –1 on any error.

long bssSeek(bssNav *nav, time_t time, time_t *curTime, unsigned long *count)
Sets the current playback position in the database, in *nav*, to the data received in the bundle with the earliest creation time that was greater than or equal to *time*. Populates *nav* and also returns the creation time and bundle ID count of that bundle in *curTime* and *count*. Returns the length of data at this location, or –1 on any error.

long bssSeek_read(bssNav *nav, time_t time, time_t *curTime, unsigned long *count, char *data, int dataLen)
A convenience function that performs *bssSeek()* followed by an immediate *bssRead()* to return the data at the new playback position. Returns the length of data read, or –1 on any error.

long bssNext(bssNav *nav, time_t *curTime, unsigned long *count)
Sets the playback position in the database, in *nav*, to the data received in the bundle with the earliest creation time and ID count greater than that of the bundle at the current playback position. Populates *nav* and also returns the creation time and bundle ID count of that bundle in *curTime* and *count*. Returns the length of data at this location (if any), –2 on reaching end of list, or –1 on any error.

long bssNext_read(bssNav *nav, time_t *curTime, unsigned long *count, char *data, int dataLen)
A convenience function that performs *bssNext()* followed by an immediate *bssRead()* to return the data at the new playback position. Returns the length of data read, –2 on reaching end of list, or –1 on any error.

long bssPrev(bssNav *nav, time_t *curTime, unsigned long *count)
Sets the playback position in the database, in *nav*, to the data received in the bundle with the latest creation time and ID count earlier than that of the bundle at the current playback position. Populates *nav* and also returns the creation time and bundle ID count of that bundle in *curTime* and *count*. Returns the length of data at this location (if any), –2 on reaching end of list, or –1 on any error.

long bssPrev_read(bssNav *nav, time_t *curTime, unsigned long *count, char *data, int dataLen)
A convenience function that performs *bssPrev()* followed by an immediate *bssRead()* to return the data at the new playback position. Returns the length of data read, –2 on reaching end of list, or –1 on any error.

SEE ALSO

bp (3)

NAME

cfdp – CCSDS File Delivery Protocol (CFDP) communications library

SYNOPSIS

```
#include "cfdp.h"

typedef int (*CfdpReaderFn)(int fd, unsigned int *checksum);

typedef enum
{
    CfdpCreateFile = 0,
    CfdpDeleteFile,
    CfdpRenameFile,
    CfdpAppendFile,
    CfdpReplaceFile,
    CfdpCreateDirectory,
    CfdpRemoveDirectory,
    CfdpDenyFile,
    CfdpDenyDirectory
} CfdpAction;

typedef enum
{
    CfdpNoEvent = 0,
    CfdpTransactionInd,
    CfdpEofSentInd,
    CfdpTransactionFinishedInd,
    CfdpMetadataRecvInd,
    CfdpFileSegmentRecvInd,
    CfdpEofRecvInd,
    CfdpSuspendedInd,
    CfdpResumedInd,
    CfdpReportInd,
    CfdpFaultInd,
    CfdpAbandonedInd
} CfdpEventType;

[see description for available functions]
```

DESCRIPTION

The cfdp library provides functions enabling application software to use CFDP to send and receive files. It conforms to the Class 1 (Unacknowledged) service class defined in the CFDP Blue Book.

In the ION implementation of CFDP, the CFDP notion of **entity ID** is taken to be identical to the BP (CBHE) notion of DTN **node number**.

CFDP entity and transaction numbers may be up to 64 bits in length. For portability to 32−bit machines, these numbers are stored in the CFDP state machine as structures of type CfdpNumber.

To simplify the interface between CFDP the user application without risking storage leaks, the CFDP-ION API uses MetadataList objects. A MetadataList is a specially formatted SDR list of user messages, filestore requests, or filestore responses. During the time that a MetadataList is pending processing via the CFDP API, but is not yet (or is no longer) reachable from any FDU object, a pointer to the list is appended to one of the lists of MetadataList objects in the CFDP non-volatile database. This assures that any unplanned termination of the CFDP daemons won't leave any SDR lists unreachable — and therefore un-recyclable — due to the absence of references to those lists. Restarting CFDP automatically purges any unused MetadataLists from the CFDP database. The ''user data'' variable of the MetadataList itself is used to

implement this feature: while the list is reachable only from the database root, its user data variable points to the database root list from which it is referenced; while the list is attached to a File Delivery Unit, its user data is null.

By default, CFDP transmits the data in a source file in segments of fixed size. The user application can override this behavior at the time transmission of a file is requested, by supplying a file reader callback function that reads the file — one byte at a time — until it detects the end of a ''record'' that has application significance. Each time CFDP calls the reader function, the function must return the length of one such record (which must be no greater than 65535).

When CFDP is used to transmit a file, a 32–bit checksum must be provided in the ''EOF'' PDU to enable the receiver of the file to assure that it was not corrupted in transit. When an application-specific file reader function is supplied, that function is responsible for updating the computed checksum as it reads each byte of the file; a CFDP library function is provided for this purpose.

The return value for each CFDP ''request'' function (put, cancel, suspend, resume, report) is a reference number that enables ''events'' obtained by calling *cfdp_get_event()* to be matched to the requests that caused them. Events with reference number set to zero are events that were caused by autonomous CFDP activity, e.g., the reception of a file data segment.

int *cfdp_attach()*
: Attaches the application to CFDP functionality on the local computer. Returns 0 on success, –1 on any error.

int *cfdp_entity_is_started()*
: Returns 1 if the local CFDP entity has been started and not yet stopped, 0 otherwise.

void *cfdp_detach()*
: Terminates all access to CFDP functionality on the local computer.

void cfdp_compress_number(CfdpNumber *toNbr, uvast from)
: Converts an unsigned **vast** number into a CfdpNumber structure, e.g., for use when invoking the *cfdp_put()* function.

void cfdp_decompress_number(uvast toNbr, CfdpNumber *from)
: Converts a numeric value in a CfdpNumber structure to an unsigned **vast** integer.

void cfdp_update_checksum(unsigned char octet, unsigned int *offset, unsigned int *checksum)
: For use by an application-specific file reader callback function, which must pass to *cfdp_update_checksum()* the value of each byte (octet) it reads. *offset* must be *octet*'s displacement in bytes from the start of the file. The *checksum* pointer is provided to the reader function by CFDP.

MetadataList *cfdp_create_usrmsg_list()*
: Creates a non-volatile linked list, suitable for containing messages-to-user that are to be presented to *cfdp_put()*.

int cfdp_add_usrmsg(MetadataList list, unsigned char *text, int length)
: Appends the indicated message-to-user to *list*.

int cfdp_get_usrmsg(MetadataList list, unsigned char *textBuf, int *length)
: Removes from *list* the first of the remaining messages-to-user contained in the list and delivers its text and length. When the last message in the list is delivered, destroys the list.

void cfdp_destroy_usrmsg_list(MetadataList *list)
: Removes and destroys all messages-to-user in *list* and destroys the list.

MetadataList *cfdp_create_fsreq_list()*
: Creates a non-volatile linked list, suitable for containing filestore requests that are to be presented to *cfdp_put()*.

int cfdp_add_fsreq(MetadataList list, CfdpAction action, char *firstFileName, char *seconfdFIleName)
: Appends the indicated filestore request to *list*.

int cfdp_get_fsreq(MetadataList list, CfdpAction *action, char *firstFileNameBuf, char *secondFileNameBuf)

Removes from *list* the first of the remaining filestore requests contained in the list and delivers its action code and file names. When the last request in the list is delivered, destroys the list.

void cfdp_destroy_fsreq_list(MetadataList *list)

Removes and destroys all filestore requests in *list* and destroys the list.

int cfdp_get_fsresp(MetadataList list, CfdpAction *action, int *status, char *firstFileNameBuf, char *secondFileNameBuf, char *messageBuf)

Removes from *list* the first of the remaining filestore responses contained in the list and delivers its action code, status, file names, and message. When the last response in the list is delivered, destroys the list.

void cfdp_destroy_fsresp_list(MetadataList *list)

Removes and destroys all filestore responses in *list* and destroys the list.

int cfdp_read_space_packets(int fd, unsigned int *checksum)

This is a standard ''reader'' function that segments the source file on CCSDS space packet boundaries. Multiple small packets may be aggregated into a single file data segment.

int cfdp_read_text_lines(int fd, unsigned int *checksum)

This is a standard ''reader'' function that segments a source file of text lines on line boundaries.

int cfdp_put(CfdpNumber *destinationEntityNbr, unsigned int utParmsLength, unsigned char *utParms, char *sourceFileName, char *destFileName, CfdpReaderFn readerFn, CfdpHandler *faultHandlers, unsigned int flowLabelLength, unsigned char *flowLabel, MetadataList messagesToUser, MetadataList filestoreRequests, CfdpTransactionId *transactionId)

Sends the file identified by *sourceFileName* to the CFDP entity identified by *destinationEntityNbr*. *destinationFileName* is used to indicate the name by which the file will be catalogued upon arrival at its final destination; if NULL, the destination file name defaults to *sourceFileName*. If *sourceFileName* is NULL, it is assumed that the application is requesting transmission of metadata only (as discussed below) and *destinationFileName* is ignored. Note that both *sourceFileName* and *destinationFileName* are interpreted as path names, i.e., directory paths may be indicated in either or both. The syntax of path names is opaque to CFDP; the syntax of *sourceFileName* must conform to the path naming syntax of the source entity's file system and the syntax of *destinationFileName* must conform to the path naming syntax of the destination entity's file system.

The byte array identified by *utParms*, if non-NULL, is interpreted as transmission control information that is to be passed on to the UT layer. The nominal UT layer for ION's CFDP being Bundle Protocol, the *utParms* array is normally a pointer to a structure of type BpUtParms; see the *bp* man page for a discussion of the parameters in that structure.

messagesToUser and *filestoreRequests*, where non-zero, must be the addresses of non-volatile linked lists (that is, linked lists in ION's SDR database) of CfdpMsgToUser and CfdpFilestoreRequest objects identifying metadata that are intended to accompany the transmitted file. Note that this metadata may accompany a file of zero length (as when *sourceFileName* is NULL as noted above) — a transmission of metadata only.

On success, the function populates **transactionID* with the source entity ID and the transaction number assigned to this transmission and returns the request number identifying this ''put'' request. The transaction ID may be used to suspend, resume, cancel, or request a report on the progress of this transmission. *cfdp_put()* returns –1 on any error.

int cfdp_cancel(CfdpTransactionId *transactionId)

Cancels transmission or reception of the indicated transaction. Note that, since the ION implementation of CFDP is Unacknowledged, cancellation of a file transmission may have little effect. Returns request number on success, –1 on any error.

int cfdp_suspend(CfdpTransactionId *transactionId)
: Suspends transmission of the indicated transaction. Note that, since the ION implementation of CFDP is Unacknowledged, suspension of a file transmission may have little effect. Returns request number on success, −1 on any error.

int cfdp_resume(CfdpTransactionId *transactionId)
: Resumes transmission of the indicated transaction. Note that, since the ION implementation of CFDP is Unacknowledged, resumption of a file transmission may have little effect. Returns request number on success, −1 on any error.

int cfdp_report(CfdpTransactionId *transactionId)
: Requests issuance of a report on the transmission or reception progress of the indicated transaction. The report takes the form of a character string that is returned in a CfdpEvent structure; use *cfdp_get_event()* to receive the event (which may be matched to the request by request number). Returns request number on success, 0 if transaction is unknown, −1 on any error.

int cfdp_get_event(CfdpEventType *type, time_t *time, int *reqNbr, CfdpTransactionId *transactionId, char *sourceFileNameBuf, char *destFileNameBuf, unsigned int *fileSize, MetadataList *messagesToUser, unsigned int *offset, unsigned int *length, CfdpCondition *condition, unsigned int *progress, CfdpFileStatus *fileStatus, CfdpDeliveryCode *deliveryCode, CfdpTransactionId *originatingTransactionId, char *statusReportBuf, MetadataList *filestoreResponses);
: Populates return value fields with data from the oldest CFDP event not yet delivered to the application.

  cfdp_get_event() always blocks indefinitely until an CFDP processing event is delivered or the function is interrupted by an invocation of *cfdp_interrupt()*.

  On application error, returns zero but sets errno to EINVAL. Returns −1 on system failure, zero otherwise.

void *cfdp_interrupt()*
: Interrupts an *cfdp_get_event()* invocation. This function is designed to be called from a signal handler.

int cfdp_preview(CfdpTransactionId *transactionId, unsigned int offset, int length, char *buffer);
: This function is provided to enable the application to get an advance look at the content of a file that CFDP has not yet fully received. Reads *length* bytes starting at *offset* bytes from the start of the file that is the destination file of the transaction identified by *transactionID*, into *buffer*. On user error (transaction is nonexistent or is outbound, or offset is beyond the end of file) returns 0. On system failure, returns −1. Otherwise returns number of bytes read.

int cfdp_map(CfdpTransactionId *transactionId, unsigned int *extentCount, CfdpExtent *extentsArray);
: This function is provided to enable the application to report on the portions of a partially-received file that have been received and written. Lists the received continuous data extents in the destination file of the transaction identified by *transactionID*. The extents (offset and length) are returned in the elements of *extentsArray*; the number of extents returned in the array is the total number of continuous extents received so far, or *extentCount*, whichever is less. The total number of extents received so far is returned as the new value of *extentCount*. On system failure, returns −1. Otherwise returns 0.

SEE ALSO

cfdpadmin (1), *cfdprc* (5)

NAME

dgr – Datagram Retransmission system library

SYNOPSIS

```
#include "dgr.h"

[see description for available functions]
```

DESCRIPTION

The DGR library is an alternative implementation of a subset of LTP, intended for use over UDP/IP in the Internet; unlike ION's canonical LTP implementation it includes a congestion control mechanism that interprets LTP block transmission failure as an indication of network congestion (not data corruption) and reduces data transmission rate in response.

As such, DGR differs from many reliable-UDP systems in two main ways:

```
It uses adaptive timeout interval computation techniques
borrowed from TCP to try to avoid introducing congestion
into the network.

It borrows the concurrent-session model of transmission
from LTP (and ultimately from CFDP), rather than waiting
for one datagram to be acknowledged before sending the next,
to improve bandwidth utilization.
```

At this time DGR is interoperable with other implementations of LTP only when each block it receives is transmitted in a single LTP data segment encapsulated in a single UDP datagram. More complex LTP behavior may be implemented in the future.

int dgr_open(uvast ownEngineId, unsigned in clientSvcId, unsigned short ownPortNbr, unsigned int ownIpAddress, char *memmgrName, Dgr *dgr, DgrRC *rc)

Establishes the application's access to DGR communication service.

ownEngineId is the sending LTP engine ID that will characterize segments issued by this DGR service access point. In order to prevent erroneous system behavior, never assign the same LTP engine ID to any two interoperating DGR SAPs.

clientSvcId identifies the LTP client service to which all LTP segments issued by this DGR service access point will be directed.

ownPortNbr is the port number to use for DGR service. If zero, a system-assigned UDP port number is used.

ownIpAddress is the Internet address of the network interface to use for DGR service. If zero, this argument defaults to the address of the interface identified by the local machine's host name.

memmgrName is the name of the memory manager (see *memmgr*(3)) to use for dynamic memory management in DGR. If NULL, defaults to the standard system *malloc()* and *free()* functions.

dgr is the location in which to store the service access pointer that must be supplied on subsequent DGR function invocations.

rc is the location in which to store the DGR return code resulting from the attempt to open this service access point (always DgrOpened).

On any failure, returns –1. On success, returns zero.

void dgr_getsockname(Dgr dgr, unsigned short *portNbr, unsigned int *ipAddress)

States the port number and IP address of the UDP socket used for this DGR service access point.

void dgr_close(Dgr dgr)

Reverses *dgr_open()*, releasing resources where possible.

int dgr_send(Dgr dgr, unsigned short toPortNbr, unsigned int toIpAddress, int notificationFlags, char *content, int length, DgrRC *rc)

Sends the indicated content, of length as indicated, to the remote DGR service access point identified by *toPortNbr* and *toIpAddress*. The message will be retransmitted as necessary until either it is acknowledged or DGR determines that it cannot be delivered.

notificationFlags, if non-zero, is the logical OR of the notification behaviors requested for this datagram. Available behaviors are DGR_NOTE_FAILED (a notice of datagram delivery failure will issued if delivery of the datagram fails) and DGR_NOTE_ACKED (a notice of datagram delivery success will be issued if delivery of the datagram succeeds). Notices are issued via *dgr_receive()* that is, the thread that calls *dgr_receive()* on this DGR service access point will receive these notices interspersed with inbound datagram contents.

length of content must be greater than zero and may be as great as 65535, but lengths greater than 8192 may not be supported by the local underlying UDP implementation; to minimize the chance of data loss when transmitting over the internet, length should not exceed 512.

rc is the location in which to store the DGR return code resulting from the attempt to send the content.

On any failure, returns −1 and sets **rc* to DgrFailed. On success, returns zero.

int dgr_receive(Dgr dgr, unsigned short *fromPortNbr, unsigned int *fromIpAddress, char *content, int *length, int *errnbr, int timeoutSeconds, DgrRC *rc)

Delivers the oldest undelivered DGR event queued for delivery.

DGR events are of two type: (a) messages received from a remote DGR service access point and (b) notices of previously sent messages that DGR has determined either have been or cannot be delivered, as requested in the *notificationFlags* parameters provided to the *dgr_send()* calls that sent those messages.

In the former case, *dgr_receive()* will place the content of the inbound message in *content*, its length in *length*, and the IP address and port number of the sender in *fromIpAddress* and *fromPortNbr*, and it will set **rc* to DgrDatagramReceived and return zero.

In the latter case, *dgr_receive()* will place the content of the affected **outbound** message in *content* and its length in *length* and return zero. If the event being reported is a delivery success, then DgrDatagramAcknowledged will be placed in **rc*. Otherwise, DgrDatagramNotAcknowledged will be placed in **rc* and the relevant errno (if any) will be placed in **errnbr*.

The *content* buffer should be at least 65535 bytes in length to enable delivery of the content of the received or delivered/undeliverable message.

timeoutSeconds controls blocking behavior. If *timeoutSeconds* is DGR_BLOCKING (i.e., −1), *dgr_receive()* will not return until (a) there is either an inbound message to deliver or an outbound message delivery result to report, or (b) the function is interrupted by means of *dgr_interrupt()*. If *timeoutSeconds* is DGR_POLL (i.e., zero), *dgr_receive()* returns immediately; if there is currently no inbound message to deliver and no outbound message delivery result to report, the function sets **rc* to DgrTimedOut and returns zero. For any other positive value of *timeoutSeconds*, *dgr_receive()* returns after the indicated number of seconds have lapsed (in which case the returned value of **rc* is DgrTimedOut), or when there is a message to deliver or a delivery result to report, or when the function is interrupted by means of *dgr_interrupt()*, whichever occurs first. When the function returns due to interruption by *dgr_interrupt()*, the value placed in **rc* is DgrInterrupted instead of DgrDatagramReceived.

rc is the location in which to store the DGR return code resulting from the attempt to receive content.

On any I/O error or other unrecoverable system error, returns −1. Otherwise always returns zero, placing DgrFailed in **rc* and writing a failure message in the event of an operating error.

void dgr_interrupt(Dgr dgr)

Interrupts a *dgr_receive()* invocation that is currently blocked. Designed to be called from a signal handler; for this purpose, *dgr* may need to be obtained from a static variable.

SEE ALSO

ltp (3), *file2dgr* (1), *dgr2file* (1)

NAME

ion – Interplanetary Overlay Network common definitions and functions

SYNOPSIS

```
#include "ion.h"

[see description for available functions]
```

DESCRIPTION

The Interplanetary Overlay Network (ION) software distribution is an implementation of Delay-Tolerant Networking (DTN) architecture as described in Internet RFC 4838. It is designed to enable inexpensive insertion of DTN functionality into embedded systems such as robotic spacecraft. The intent of ION deployment in space flight mission systems is to reduce cost and risk in mission communications by simplifying the construction and operation of automated digital data communication networks spanning space links, planetary surface links, and terrestrial links.

The ION distribution comprises the following software packages:

ici (Interplanetary Communication Infrastructure), a set of general-purpose libraries providing common functionality to the other packages.

ltp (Licklider Transmission Protocol), a core DTN protocol that provides transmission reliability based on delay-tolerant acknowledgments, timeouts, and retransmissions.

bp (Bundle Protocol), a core DTN protocol that provides delay-tolerant forwarding of data through a network in which continuous end-to-end connectivity is never assured, including support for delay-tolerant dynamic routing. The Bundle Protocol (BP) specification is defined in Internet RFC 5050.

dgr (Datagram Retransmission), a library that enables data to be transmitted via UDP with reliability comparable to that provided by TCP. DGR is an alternative implementation of LTP, designed for use within an internet.

ams (Asynchronous Message Service), *cfdp* (CCSDS File Delivery Protocol), and *bss* (Bundle Streaming Service), application-layer services that are not part of the DTN architecture but utilize underlying DTN protocols.

Taken together, the packages included in the ION software distribution constitute a communication capability characterized by the following operational features:

Reliable conveyance of data over a DTN, i.e., a network in which it might never be possible for any node to have reliable information about the detailed current state of any other node.

Built on this capability, reliable distribution of short messages to multiple recipients (subscribers) residing in such a network.

Management of traffic through such a network.

Facilities for monitoring the performance of the network.

Robustness against node failure.

Portability across heterogeneous computing platforms.

High speed with low overhead.

Easy integration with heterogeneous underlying communication infrastructure, ranging from Internet to dedicated spacecraft communication links.

While most of the ici package consists of libraries providing functionality that may be of general utility in any complex embedded software system, the functions and macros described below are specifically designed to support operations of ION's delay-tolerant networking protocol stack.

TIMESTAMPBUFSZ

This macro returns the recommended size of a buffer that is intended to contain a timestamp in ION-standard format:

yyyy/mm/dd−hh:mm:ss

int *ionAttach()*

Attaches the invoking task to ION infrastructure as previously established by running the *ionadmin* utility program on the same computer. Returns zero on success, −1 on any error.

void *ionDetach()*

Detaches the invoking task from ION infrastructure. In particular, releases handle allocated for access to ION's non-volatile database. **NOTE**, though, that *ionDetach()* has no effect when the invoking task is running in a non-memory-protected environment, such as VxWorks, where all ION resource access variables are shared by all tasks: no single task could detach without crashing all other ION tasks.

void ionProd(uvast fromNode, uvast toNode, unsigned int xmitRate, unsigned int owlt)

This function is designed to be called from an operating environment command or a fault protection routine, to enable operation of a node to resume when all of its scheduled contacts are in the past (making it impossible to use a DTN communication contact to assert additional future communication contacts). The function asserts a single new unidirectional contact conforming to the arguments provided, including the applicable one-way light time, with start time equal to the current time (at the moment of execution of the function) and end time equal to the start time plus 2 hours. The result of executing the function is written to the ION log using standard ION status message logging functions.

NOTE that the *ionProd()* function must be invoked twice in order to establish bidirectional communication.

void ionClear(char *srcEid, char *destEid, char *blockType)

This function is designed to be called from an operating environment command or a fault protection routine, to delete some or all of ION's Bundle Security Protocol rules when they are preventing nominal authorized operation of a node. If the first character of *blockType* is '~' then the function applies to rules for all types of BSP block; otherwise it applies only to rules for the named BSP block type: ''bab'', ''pib'', ''pcb'', or ''esb''. Only rules whose security source EIDs match *srcEid* and whose security destination EIDs match *destEid* are deleted. A rule EID matches a ''clearing'' EID if (a) every character of the clearing EID prior to the first '~' in the clearing ID (if any) is equal to the corresponding character of the rule EID **and** either the first '~' character in the clearing EID is the clearing EID's last character or else the rule EID and clearing EID are of equal length.

Object ionCreateZco(ZcoMedium source, Object location, vast offset, vast length, int *control)

This function provides a ''blocking'' implementation of admission control in ION. Like *zco_create()*, it constructs a zero-copy object (see *zco*(3)) that contains a single extent of source data residing at *location* in *source*, of which the first *offset* bytes are omitted and the next *length* bytes are included. But unlike *zco_create()*, *ionCreateZco()* will block (rather than return an immediate error indication) so long as the total amount of space in *source* that is available for new ZCO formation is less than *length*. *ionCreateZco()* returns when either (a) space has become available and the ZCO has been created, in which case the location of the ZCO is returned, or (b) the function has failed (in which case ((Object) −1) is returned), or (c) the function was interrupted by *ionCancelZcoSpaceRequest()* before space for the ZCO became available (in which case 0 is returned).

ionCreateZco() is interruptible if and only if a non-NULL value of *control* is passed to it; in that case, passing the **same** value of *control* to the *ionCancelZcoSpaceRequest()* function will interrupt *ionCreateZco()*. Note that the integer variable referenced by *control* functions as a private ION work area; its content need not be initialized — and must not be altered — by the application. Be careful when referencing a dynamic variable in *control*; static variables are safer. If *control* is NULL then *ionCreateZco()* is not interruptible by the application.

vast ionAppendZcoExtent(Object zco, ZcoMedium source, Object location, vast offset, vast length, int *control)

Similar to *ionCreateZco()* except that instead of creating a new ZCO it appends an additional extent to an existing ZCO. Returns −1 on failure, 0 on interruption by *ionCancelZcoSpaceRequest()*, *length* on success.

void ionCancelZcoSpaceRequest(int *control)

This function simply interrupts the currently blocked invocation of *ionCreateZco()* or *ionAppendZcoExtent()* that cites the same *control* value. *ionCancelZcoSpaceRequest()* is intended to be called from a signal handler, so that a signal can be used to cleanly terminate a thread that is waiting for an opportunity to create a new ZCO source data extent.

Sdr *getIonsdr()*

Returns a pointer to the SDR management object, previously acquired by calling *ionAttach()*, or zero on any error.

PsmPartition *getIonwm()*

Returns a pointer to the ION working memory partition, previously acquired by calling *ionAttach()*, or zero on any error.

int *getIonMemoryMgr()*

Returns the memory manager ID for operations on ION's working memory partition, previously acquired by calling *ionAttach()*, or –1 on any error.

int *ionLocked()*;

Returns 1 if the calling task is the owner of the current SDR transaction. Assuring that ION is locked while related critical operations are performed is essential to the avoidance of race conditions.

uvast *getOwnNodeNbr()*

Returns the Bundle Protocol node number identifying this computer, as declared when ION was initialized by *ionadmin*.

time_t *getUTCTime()*

Returns the current UTC time, as computed from local clock time and the computer's current offset from UTC (due to clock drift, **not** due to time zone difference; the **utcdelta**) as managed from *ionadmin*.

int *ionClockIsSynchronized()*

Returns 1 if the computer on which the local ION node is running has a synchronized clock , i.e., a clock that reports the current UTC time as a value that differs from the correct time by an interval approximately equal to the currently asserted offset from UTC due to clock drift; returns zero otherwise.

If the machine's clock is synchronized then its reported values (as returned by *getUTCTime()*) can safely be used as the creation times of new bundles and the expiration time of such a bundle can accurately be computed as the sum of the bundle's creation time and time to live. If not, then the creation timestamp time of new bundles sourced at the local ION node must be zero and the creation timestamp sequence numbers must increase monotonically forever, never rolling over to zero.

void writeTimestampLocal(time_t timestamp, char *timestampBuffer)

Expresses the time value in *timestamp* as a local timestamp string in ION-standard format, as noted above, in *timestampBuffer*.

void writeTimestampUTC(time_t timestamp, char *timestampBuffer)

Expresses the time value in *timestamp* as a UTC timestamp string in ION-standard format, as noted above, in *timestampBuffer*.

time_t readTimestampLocal(char *timestampBuffer, time_t referenceTime)

Parses the local timestamp string in *timestampBuffer* and returns the corresponding time value (as would be returned by *time* (2)), or zero if the timestamp string cannot be parsed successfully. The timestamp string is normally expected to be an absolute expression of local time in ION-standard format as noted above. However, a relative time expression variant is also supported: if the first character of *timestampBuffer* is '+' then the remainder of the string is interpreted as a count of seconds; the sum of this value and the time value in *referenceTime* is returned.

time_t readTimestampUTC(char *timestampBuffer, time_t referenceTime)

Same as *readTimestampLocal()* except that if *timestampBuffer* is not a relative time expression then it is interpreted as an absolute expression of UTC time in ION-standard format as noted above.

STATUS MESSAGES

ION uses *writeMemo()*, *putErrmsg()*, and *putSysErrmsg()* to log several different types of standardized status messages.

Informational messages

These messages are generated to inform the user of the occurrence of events that are nominal but significant, such as the controlled termination of a daemon or the production of a congestion forecast. Each informational message has the following format:

{*yyyy/mm/dd hh:mm:ss*} [i] *text*

Warning messages

These messages are generated to inform the user of the occurrence of events that are off-nominal but are likely caused by configuration or operational errors rather than software failure. Each warning message has the following format:

{*yyyy/mm/dd hh:mm:ss*} [?] *text*

Diagnostic messages

These messages are produced by calling *putErrmsg()* or *putSysErrmsg()*. They are generated to inform the user of the occurrence of events that are off-nominal and might be due to errors in software. The location within the ION software at which the off-nominal condition was detected is indicated in the message:

{*yyyy/mm/dd hh:mm:ss*} at line *nnn* of *sourcefilename*, *text* (*argument*)

Note that the *argument* portion of the message (including its enclosing parentheses) will be provided only when an argument value seems potentially helpful in fault analysis.

Bundle Status Report (BSR) messages

A BSR message informs the user of the arrival of a BSR, a Bundle Protocol report on the status of some bundle. BSRs are issued in the course of processing bundles for which one or more status report request flags are set, and they are also issued when bundles for which custody transfer is requested are destroyed prior to delivery to their destination endpoints. A BSR message is generated by **ipnadminep** upon reception of a BSR. The time and place (node) at which the BSR was issued are indicated in the message:

{*yyyy/mm/dd hh:mm:ss*} [s] (*sourceEID*)/*creationTimeSeconds*:*counter*/*fragmentOffset* status *flagsByte* at *time* on *endpointID*, '*reasonString*'.

Communication statistics messages

A network performance report is a set of eight communication statistics messages, one for each of eight different types of network activity. A report is issued every time contact transmission or reception starts or stops, except when there is no activity of any kind on the local node since the prior report. When a report is issued, statistic messages are generated to summarize all network activity detected since the prior report, after which all network activity counters and accumulators are reset to zero.

NOTE also that the **bpstats** utility program can be invoked to issue an interim network performance report at any time. Issuance of interim status reports does **not** cause network activity counters and accumulators to be reset to zero.

Statistics messages have the following format:

{*yyyy/mm/dd hh:mm:ss*} [x] *xxx* from *ttttttt* to *TTTTTTTT*: (0) *aaaa bbbbbbbbbb* (1) *cccc ddddddddd* (2) *eeee fffffffff* (+) *gggg hhhhhhhhhh*

xxx indicates the type of network activity that the message is reporting on. Statistics for eight different types of network activity are reported:

src This message reports on the bundles sourced at the local node during the indicated interval.

fwd
This message reports on the bundles forwarded by the local node. When a bundle is re-forwarded due to custody transfer timeout it is counted a second time here.

xmt
This message reports on the bundles passed to the convergence layer protocol(s) for transmission from this node. Again, a re-forwarded bundle that is then re-transmitted at the convergence layer is counted a second time here.

rcv This message reports on the bundles from other nodes that were received at the local node.

dlv This message reports on the bundles delivered to applications via endpoints on the local node.

ctr This message reports on the custody refusal signals received at the local node.

rfw This message reports on bundles for which convergence-layer transmission failed at this node, causing the bundles to be re-forwarded.

exp This message reports on the bundles destroyed at this node due to TTL expiration.

ttttttt and *TTTTTTTT* indicate the start and end times of the interval for which statistics are being reported, expressed in *yyyy/mm/dd−hh:mm:ss* format. *TTTTTTTT* is the current time and *ttttttt* is the time of the prior report.

Each of the four value pairs following the colon (:) reports on the number of bundles counted for the indicated type of network activity, for the indicated traffic flow, followed by the sum of the sizes of the payloads of all those bundles. The four traffic flows for which statistics are reported are ‘‘(0)’’ the priority−0 or ‘‘bulk’’ traffic, ‘‘(1)’’ the priority−1 ‘‘standard’’ traffic, ‘‘(2)’’ the priority−2 ‘‘expedited’’ traffic, and ‘‘(+)’’ the total for all classes of service.

Free-form messages
Other status messages are free-form, except that date and time are always noted just as for the documented status message types.

SEE ALSO

ionadmin (1), *rfxclock* (1), *bpstats* (1), *llcv* (3), *lyst* (3), *memmgr* (3), *platform* (3), *psm* (3), *sdr* (3), *zco* (3), *ltp* (3), *bp* (3), *cfdp* (3), *ams* (3), *bss* (3)

NAME

llcv – library for manipulating linked–list condition variable objects

SYNOPSIS

```
#include "llcv.h"

typedef struct llcv_str
{
    Lyst            list;
    pthread_mutex_t mutex;
    pthread_cond_t  cv;
} *Llcv;

typedef int (*LlcvPredicate)(Llcv);

[see description for available functions]
```

DESCRIPTION

A ‘‘linked-list condition variable’’ object (LLCV) is an inter-thread communication mechanism that pairs a process-private linked list in memory with a condition variable as provided by the pthreads library. LLCVs echo in thread programming the standard ION inter-process or inter-task communication model that pairs shared-memory semaphores with linked lists in shared memory or shared non-volatile storage. As in the semaphore/list model, variable-length messages may be transmitted; the resources allocated to the communication mechanism grow and shrink to accommodate changes in data rate; the rate at which messages are issued is completely decoupled from the rate at which messages are received and processed. That is, there is no flow control, no blocking, and therefore no possibility of deadlock or ‘‘deadly embrace’’. Traffic spikes are handled without impact on processing rate, provided sufficient memory is provided to accommodate the peak backlog.

An LLCV comprises a Lyst, a mutex, and a condition variable. The Lyst may be in either private or shared memory, but the Lyst itself is not shared with other processes. The reader thread waits on the condition variable until signaled by a writer that some condition is now true. The standard Lyst API functions are used to populate and drain the linked list. In order to protect linked list integrity, each thread must call *llcv_lock()* before operating on the Lyst and *llcv_unlock()* afterwards. The other llcv functions merely effect flow signaling in a way that makes it unnecessary for the reader to poll or busy-wait on the Lyst.

Llcv llcv_open(Lyst list, Llcv llcv)

Opens an LLCV, initializing as necessary. The *list* argument must point to an existing Lyst, which may reside in either private or shared dynamic memory. *llcv* must point to an existing llcv_str management object, which may reside in either static or dynamic (private or shared) memory — but *NOT* in stack space. Returns *llcv* on success, NULL on any error.

void llcv_lock(Llcv llcv)

Locks the LLCV’s Lyst so that it may be updated or examined safely by the calling thread. Fails silently on any error.

void llcv_unlock(Llcv llcv)

Unlocks the LLCV’s Lyst so that another thread may lock and update or examine it. Fails silently on any error.

int llcv_wait(Llcv llcv, LlcvPredicate cond, int microseconds)

Returns when the Lyst encapsulated within the LLCV meets the indicated condition. If *microseconds* is non-negative, will return –1 and set *errno* to ETIMEDOUT when the indicated number of microseconds has passed, if and only if the indicated condition has not been met by that time. Negative values of the microseconds argument other than LLCV_BLOCKING (defined as –1) are illegal. Returns –1 on any error.

void llcv_signal(Llcv llcv, LlcvPredicate cond)
: Locks the indicated LLCV's Lyst; tests (evaluates) the indicated condition with regard to that LLCV; if the condition is true, signals to the waiting reader on this LLCV (if any) that the Lyst encapsulated in the indicated LLCV now meets the indicated condition; and unlocks the Lyst.

void llcv_signal_while_locked(Llcv llcv, LlcvPredicate cond)
: Same as *llcv_signal()* except does not lock the Llcv's mutex before signalling or unlock afterwards. For use when the Llcv is already locked; prevents deadlock.

void llcv_close(Llcv llcv)
: Destroys the indicated LLCV's mutex and condition variable. Fails silently (and has no effect) if a reader is currently waiting on the Llcv.

int llcv_lyst_is_empty(Llcv Llcv)
: A built-in "convenience" predicate, for use when calling *llcv_wait()*, *llcv_signal()*, or *llcv_signal_while_locked()*. Returns true if the length of the indicated LLCV's encapsulated Lyst is zero, false otherwise.

int llcv_lyst_not_empty(Llcv Llcv)
: A built-in "convenience" predicate, for use when calling *llcv_wait()*, *llcv_signal()*, or *llcv_signal_while_locked()*. Returns true if the length of the LLCV's encapsulated Lyst is non-zero, false otherwise.

SEE ALSO

lyst (3)

NAME

lyst – library for manipulating generalized doubly linked lists

SYNOPSIS

```
#include "lyst.h"

typedef int  (*LystCompareFn)(void *s1, void *s2);
typedef void (*LystCallback)(LystElt elt, void *userdata);

[see description for available functions]
```

DESCRIPTION

The "lyst" library uses two types of objects, *Lyst* objects and *LystElt* objects. A Lyst knows how many elements it contains, its first and last elements, the memory manager used to create/destroy the Lyst and its elements, and how the elements are sorted. A LystElt knows its content (normally a pointer to an item in memory), what Lyst it belongs to, and the LystElts before and after it in that Lyst.

Lyst lyst_create(void)
: Create and return a new Lyst object without any elements in it. All operations performed on this Lyst will use the allocation/deallocation functions of the default memory manager "std" (see *memmgr* (3)). Returns NULL on any failure.

Lyst lyst_create_using(unsigned memmgrId)
: Create and return a new Lyst object without any elements in it. All operations performed on this Lyst will use the allocation/deallocation functions of the specified memory manager (see *memmgr* (3)). Returns NULL on any failure.

void lyst_clear(Lyst list)
: Clear a Lyst, i.e. free all elements of *list*, calling the Lyst's deletion function if defined, but without destroying the Lyst itself.

void lyst_destroy(Lyst list)
: Destroy a Lyst. Will free all elements of *list*, calling the Lyst's deletion function if defined.

void lyst_compare_set(Lyst list, LystCompareFn compareFn)
LystCompareFn lyst_compare_get(Lyst list)
: Set/get comparison function for specified Lyst. Comparison functions are called with two Lyst element data pointers, and must return a negative integer if first is less than second, 0 if both are equal, and a positive integer if first is greater than second (i.e., same return values as *strcmp* (3)). The comparison function is used by the *lyst_insert()*, *lyst_search()*, *lyst_sort()*, and *lyst_sorted()* functions.

void lyst_direction_set(Lyst list, LystSortDirection direction)
: Set sort direction (either LIST_SORT_ASCENDING or LIST_SORT_DESCENDING) for specified Lyst. If no comparison function is set, then this controls whether new elements are added to the end or beginning (respectively) of the Lyst when *lyst_insert()* is called.

void lyst_delete_set(Lyst list, LystCallback deleteFn, void *userdata)
: Set user deletion function for specified Lyst. This function is automatically called whenever an element of the Lyst is deleted, to perform any user-required processing. When automatically called, the deletion function is passed two arguments: the element being deleted and the *userdata* pointer specified in the *lyst_delete_set()* call.

void lyst_insert_set(Lyst list, LystCallback insertFn, void *userdata)
: Set user insertion function for specified Lyst. This function is automatically called whenever a Lyst element is inserted into the Lyst, to perform any user-required processing. When automatically called, the insertion function is passed two arguments: the element being inserted and the *userdata* pointer specified in the *lyst_insert_set()* call.

unsigned long lyst_length(Lyst list)
: Return the number of elements in the Lyst.

LystElt lyst_insert(Lyst list, void *data)

Create a new element whose content is the pointer value *data* and insert it into the Lyst. Uses the Lyst's comparison function to select insertion point, if defined; otherwise adds the new element at the beginning or end of the Lyst, depending on the Lyst sort direction setting. Returns a pointer to the newly created element, or NULL on any failure.

LystElt lyst_insert_first(Lyst list, void *data)
LystElt lyst_insert_last(Lyst list, void *data)

Create a new element and insert it at the beginning/end of the Lyst. If these functions are used when inserting elements into a Lyst with a defined comparison function, then the Lyst may get out of order and future calls to *lyst_insert()* can put new elements in unpredictable locations. Returns a pointer to the newly created element, or NULL on any failure.

LystElt lyst_insert_before(LystElt element, void *data)
LystElt lyst_insert_after(LystElt element, void *data)

Create a new element and insert it before/after the specified element. If these functions are used when inserting elements into a Lyst with a defined comparison function, then the Lyst may get out of order and future calls to lyst_insert can put new elements in unpredictable locations. Returns a pointer to the newly created element, or NULL on any failure.

void lyst_delete(LystElt element)

Delete the specified element from its Lyst and deallocate its memory. Calls the user delete function if defined.

LystElt lyst_first(Lyst list)
LystElt lyst_last(Lyst list)

Return a pointer to the first/last element of a Lyst.

LystElt lyst_next(LystElt element)
LystElt lyst_prev(LystElt element)

Return a pointer to the element following/preceding the specified element.

LystElt lyst_search(LystElt element, void *searchValue)

Find the first matching element in a Lyst starting with the specified element. Returns NULL if no matches are found. Uses the Lyst's comparison function if defined, otherwise searches from the given element to the end of the Lyst.

Lyst lyst_lyst(LystElt element)

Return the Lyst to which the specified element belongs.

void* lyst_data(LystElt element)
void* lyst_data_set(LystElt element, void *data)

Get/set the pointer value content of the specified Lyst element. The set routine returns the element's previous content, and the delete function is *not* called. If the *lyst_data_set()* function is used on an element of a Lyst with a defined comparison function, then the Lyst may get out of order and future calls to *lyst_insert()* can put new elements in unpredictable locations.

void lyst_sort(Lyst list)

Sort the Lyst based on the current comparison function and sort direction. A stable insertion sort is used that is very fast when the elements are already in order.

int lyst_sorted(Lyst list)

Determine whether or not the Lyst is sorted based on the current comparison function and sort direction.

void lyst_apply(Lyst list, LystCallback applyFn, void *userdata)

Apply the function *applyFn* automatically to each element in the Lyst. When automatically called, *applyFn* is passed two arguments: a pointer to an element, and the *userdata* argument specified in the call to *lyst_apply()*. *applyFn* should not delete or reorder the elements in the Lyst.

SEE ALSO

memmgr (3), *psm* (3)

NAME

memmgr – memory manager abstraction functions

SYNOPSIS

```
#include "memmgr.h"

typedef void *(* MemAllocator)
    (char *fileName, int lineNbr, size_t size);
typedef void (* MemDeallocator)
    (char *fileName, int lineNbr, void * blk);
typedef void *(* MemAtoPConverter) (unsigned int address);
typedef unsigned int (* MemPtoAConverter) (void * pointer);

unsigned int memmgr_add       (char *name,
                               MemAllocator take,
                               MemDeallocator release,
                               MemAtoPConverter AtoP,
                               MemPtoAConverter PtoA);
int memmgr_find               (char *name);
char *memmgr_name             (int mgrId);
MemAllocator memmgr_take      (int mgrId);
MemDeallocator memmgr_release (int mgrId);
MemAtoPConverter memmgr_AtoP  (int mgrId);
MemPtoAConverter memmgr_PtoA  (int mgrId;

int memmgr_open               (int memKey,
                               unsigned long memSize,
                               char **memPtr,
                               int *smId,
                               char *partitionName,
                               PsmPartition *partition,
                               int *memMgr,
                               MemAllocator afn,
                               MemDeallocator ffn,
                               MemAtoPConverter apfn,
                               MemPtoAConverter pafn);
void memmgr_destroy           (int smId,
                               PsmPartition *partition);
```

DESCRIPTION

‘‘memmgr’’ is an abstraction layer for administration of memory management. It enables multiple memory managers to coexist in a single application. Each memory manager specification is required to include pointers to a memory allocation function, a memory deallocation function, and functions for translating between local memory pointers and ‘‘addresses’’, which are abstract memory locations that have private meaning to the manager. The allocation function is expected to return a block of memory of size ‘‘size’’ (in bytes), initialized to all binary zeroes. The *fileName* and *lineNbr* arguments to the allocation and deallocation functions are expected to be the values of __FILE__ and __LINE__ at the point at which the functions are called; this supports any memory usage tracing via *sptrace* (3) that may be implemented by the underlying memory management system.

Memory managers are identified by number and by name. The identifying number for a memory manager is an index into a private, fixed-length array of up to 8 memory manager configuration structures; that is, memory manager number must be in the range 0–7. However, memory manager numbers are assigned dynamically and not always predictably. To enable multiple applications to use the same memory manager for a given segment of shared memory, a memory manager may be located by a predefined name of up to 15 characters that is known to all the applications.

The memory manager with manager number 0 is always available; its name is ‘‘std’’. Its memory allocation function is *calloc()*, its deallocation function is *free()*, and its pointer/address translation functions are merely casts.

unsigned int memmgr_add(char *name, MemAllocator take, MemDeallocator release, MemAtoPConverter AtoP, MemPtoAConverter PtoA)

Add a memory manager to the memory manager array, if not already defined; attempting to add a previously added memory manager is not considered an error. *name* is the name of the memory manager. *take* is a pointer to the manager's memory allocation function; *release* is a pointer to the manager's memory deallocation function. *AtoP* is a pointer to the manager's function for converting an address to a local memory pointer; *PtoA* is a pointer to the manager's pointer-to-address converter function. Returns the memory manager ID number assigned to the named manager, or −1 on any error.

NOTE: *memmgr_add()* is NOT thread-safe. In a multithreaded execution image (e.g., VxWorks), all memory managers should be loaded *before* any subordinate threads or tasks are spawned.

int memmgr_find(char *name)

Return the memmgr ID of the named manager, or −1 if not found.

char *memmgr_name(int mgrId)

Return the name of the manager given by *mgrId*.

MemAllocator memmgr_take(int mgrId)

Return the allocator function pointer for the manager given by *mgrId*.

memDeallocator memmgr_release(int mgrId)

Return the deallocator function pointer for the manager given by *mgrId*.

MemAtoPConverter memmgr_AtoP(int mgrId)

Return the address-to-pointer converter function pointer for the manager given by *mgrId*.

MemPtoAConverter memmgr_PtoA(int mgrId)

Return the pointer-to-address converter function pointer for the manager given by *mgrId*.

int memmgr_open(int memKey, unsigned long memSize, char **memPtr, int *smId, char *partitionName, PsmPartition *partition, int *memMgr, MemAllocator afn, MemDeallocator ffn, MemAtoPConverter apfn, MemPtoAConverter pafn);

memmgr_open() opens one avenue of access to a PSM managed region of shared memory, initializing as necessary.

In order for multiple tasks to share access to this memory region, all must cite the same *memkey* and *partitionName* when they call *memmgr_open()*. If shared access is not necessary, then *memKey* can be SM_NO_KEY and *partitionName* can be any valid partition name.

If it is known that a prior invocation of *memmgr_open()* has already initialized the region, then *memSize* can be zero and *memPtr* must be NULL. Otherwise *memSize* is required and the required value of *memPtr* depends on whether or not the memory that is to be shared and managed has already been allocated (e.g., it's a fixed region of bus memory). If so, then the memory pointer variable that *memPtr* points to must contain the address of that memory region. Otherwise, **memPtr* must contain NULL.

memmgr_open() will allocate system memory as necessary and will in any case return the address of the shared memory region in **memPtr*.

If the shared memory is newly allocated or otherwise not yet under PSM management, then *memmgr_open()* will invoke *psm_manage()* to manage the shared memory region. It will also add a catalogue for the managed shared memory region as necessary.

If *memMgr* is non-NULL, then *memmgr_open()* will additionally call *memmgr_add()* to establish a new memory manager for this managed shared memory region, as necessary. The index of the applicable memory manager will be returned in *memMgr*. If that memory manager is newly created, then the supplied *afn*, *ffn*, *apfn*, and *pafn* functions (which can be written with reference to the memory

manager index value returned in *memMgr*) have been established as the memory management functions for local private access to this managed shared memory region.

Returns 0 on success, –1 on any error.

void memmgr_destroy(int smId, PsmPartition *partition);

memmgr_destroy() terminates all access to a PSM managed region of shared memory, invoking *psm_erase()* to destroy the partition and *sm_ShmDestroy()* to destroy the shared memory object.

EXAMPLE

```
/* this example uses the calloc/free memory manager, which is
 * called "std", and is always defined in memmgr. */

#include "memmgr.h"

main()
{
    int mgrId;
    MemAllocator myalloc;
    MemDeallocator myfree;
    char *newBlock;

    mgrId = memmgr_find("std");
    myalloc = memmgr_take(mgrId);
    myfree = memmgr_release(mgrId);
    ...

    newBlock = myalloc(5000);
    ...
    myfree(newBlock);
}
```

SEE ALSO

psm (3)

NAME

platform – C software portability definitions and functions

SYNOPSIS

```
#include "platform.h"

[see description for available functions]
```

DESCRIPTION

platform is a library of functions that simplify the porting of software written in C. It provides an API that enables application code to access the resources of an abstract POSIX-compliant ‘‘least common denominator’’ operating system — typically a large subset of the resources of the actual underlying operating system.

Most of the functionality provided by the platform library is aimed at making communication code portable: common functions for shared memory, semaphores, and IP sockets are provided. The implementation of the abstract O/S API varies according to the actual operating system on which the application runs, but the API’s behavior is always the same; applications that invoke the platform library functions rather than native O/S system calls may forego some O/S–specific capability, but they gain portability at little if any cost in performance.

Differences in word size among platforms are implemented by values of the *SPACE_ORDER* macro. ‘‘Space order’’ is the base 2 log of the number of octets in a word: for 32–bit machines the space order is 2 (2^2 = 4 octets per word), for 64–bit machines it is 3 (2^3 = 8 octets per word).

A consistent platform-independent representation of large integers is useful for some applications. For this purpose, *platform* defines new types **vast** and **uvast** (unsigned vast) which are consistently defined to be 64–bit integers regardless of the platform’s native word size.

The platform.h header file #includes many of the most frequently needed header files: sys/types.h, errno.h, string.h, stdio.h, sys/socket.h, signal.h, dirent.h, netinet/in.h, unistd.h, stdlib.h, sys/time.h, sys/resource.h, malloc.h, sys/param.h, netdb.h, sys/uni.h, and fcntl.h. Beyond this, *platform* attempts to enhance compatibility by providing standard macros, type definitions, external references, or function implementations that are missing from a few supported O/S’s but supported by all others. Finally, entirely new, generic functions are provided to establish a common body of functionality that subsumes significantly different O/S–specific capabilities.

PLATFORM COMPATIBILITY PATCHES

The platform library ‘‘patches’’ the APIs of supported O/S’s to guarantee that all of the following items may be utilized by application software:

```
The strchr(), strrchr(), strcasecmp(), and strncasecmp() functions.

The unlink(), getpid(), and gettimeofday() functions.

The select() function.

The FD_BITMAP macro (used by select()).

The MAXHOSTNAMELEN macro.

The NULL macro.

The timer_t type definition.
```

PLATFORM GENERIC MACROS AND FUNCTIONS

The generic macros and functions in this section may be used in place of comparable O/S–specific functions, to enhance the portability of code. (The implementations of these macros and functions are no-ops in environments in which they are inapplicable, so they’re always safe to call.)

FDTABLE_SIZE
: The FDTABLE_SIZE macro returns the total number of file descriptors defined for the process (or VxWorks target).

ION_PATH_DELIMITER
: The ION_PATH_DELIMITER macro returns the ASCII character — either '/' or '\' — that is used as a directory name delimiter in path names for the file system used by the local platform.

oK(expression)
: The oK macro simply casts the value of *expression* to void, a way of handling function return codes that are not meaningful in this context.

CHKERR(condition)
: The CHKERR macro is an ''assert'' mechanism. It causes the calling function to return −1 immediately if *condition* is false.

CHKZERO(condition)
: The CHKZERO macro is an ''assert'' mechanism. It causes the calling function to return 0 immediately if *condition* is false.

CHKNULL(condition)
: The CHKNULL macro is an ''assert'' mechanism. It causes the calling function to return NULL immediately if *condition* is false.

CHKVOID(condition)
: The CHKVOID macro is an ''assert'' mechanism. It causes the calling function to return immediately if *condition* is false.

void snooze(unsigned int seconds)
: Suspends execution of the invoking task or process for the indicated number of seconds.

void microsnooze(unsigned int microseconds)
: Suspends execution of the invoking task or process for the indicated number of microseconds.

void getCurrentTime(struct timeval *time)
: Returns the current local time in a timeval structure (see gettimeofday(3C)).

void isprintf(char *buffer, int bufSize, char *format, ...)
: *isprintf()* is a safe, portable implementation of *snprintf()*; see the snprintf(P) man page for details. *isprintf()* differs from *snprintf()* in that it always NULL-terminates the string in *buffer*, even if the length of the composed string would equal or exceed *bufSize*. Buffer overruns are reported by log message; unlike *snprintf()*, *isprintf()* returns void.

size_t istrlen(const char *sourceString, size_t maxlen)
: *istrlen()* is a safe implementation of *strlen()*; see the *strlen*(3) man page for details. *istrlen()* differs from *strlen()* in that it takes a second argument, the maximum valid length of *sourceString*. The function returns the number of non-NULL characters in *sourceString* preceding the first NULL character in *sourceString*, provided that a NULL character appears somewhere within the first *maxlen* characters of *sourceString*; otherwise it returns *maxlen*.

char *istrcpy(char *buffer, char *sourceString, int bufSize)
: *istrcpy()* is a safe implementation of *strcpy()*; see the *strcpy*(3) man page for details. *istrcpy()* differs from *strcpy()* in that it takes a third argument, the total size of the buffer into which *sourceString* is to be copied. *istrcpy()* always NULL-terminates the string in *buffer*, even if the length of *sourceString* string would equal or exceed *bufSize* (in which case *sourceString* is truncated to fit within the buffer).

char *istrcat(char *buffer, char *sourceString, int bufSize)
: *istrcat()* is a safe implementation of *strcat()*; see the *strcat*(3) man page for details. *istrcat()* differs from *strcat()* in that it takes a third argument, the total size of the buffer for the string that is being aggregated. *istrcat()* always NULL-terminates the string in *buffer*, even if the length of *sourceString* string would equal or exceed the sum of *bufSize* and the length of the string currently occupying the buffer (in which case *sourceString* is truncated to fit within the buffer).

char *igetcwd(char *buf, size_t size)

igetcwd() is normally just a wrapper around *getcwd*(3). It differs from *getcwd*(3) only when FSWWDNAME is defined, in which case the implementation of *igetcwd()* must be supplied in an included file named ‘‘wdname.c’’; this adaptation option accommodates flight software environments in which the current working directory name must be configured rather than discovered at run time.

void isignal(int signbr, void (*handler)(int))

isignal() is a portable, simplified interface to signal handling that is functionally indistinguishable from signal(P). It assures that reception of the indicated signal will interrupt system calls in SVR4 fashion, even when running on a FreeBSD platform.

void iblock(int signbr)

iblock() simply prevents reception of the indicated signal by the calling thread. It provides a means of controlling which of the threads in a process will receive the signal cited in an invocation of *isignal()*.

char *igets(int fd, char *buffer, int buflen, int *lineLen)

igets() reads a line of text, delimited by a newline character, from *fd* into *buffer* and writes a NULL character at the end of the string. The newline character itself is omitted from the NULL-terminated text line in *buffer*; if the newline is immediately preceded by a carriage return character (i.e., the line is from a DOS text file), then the carriage return character is likewise omitted from the NULL-terminated text line in *buffer*. End of file is interpreted as an implicit newline, terminating the line. If the number of characters preceding the newline is greater than or equal to *buflen*, only the first (*buflen* − 1) characters of the line are written into *buffer*. On error the function sets **lineLen* to −1 and returns NULL. On reading end-of-file, the function sets **lineLen* to zero and returns NULL. Otherwise the function sets **lineLen* to the length of the text line in *buffer*, as if from *strlen*(3), and returns *buffer*.

int iputs(int fd, char *string)

iputs() writes to *fd* the NULL-terminated character string at *string*. No terminating newline character is appended to *string* by *iputs()*. On error the function returns −1; otherwise the function returns the length of the character string written to *fd*, as if from *strlen*(3).

vast strtovast(char *string)

Converts the leading characters of *string*, skipping leading white space and ending at the first subsequent character that can’t be interpreted as contributing to a numeric value, to a **vast** integer and returns that integer.

uvast strtouvast(char *string)

Same as *strtovast()* except the result is an unsigned **vast** integer value.

void findToken(char **cursorPtr, char **token)

Locates the next non-whitespace lexical token in a character array, starting at **cursorPtr*. The function NULL-terminates that token within the array and places a pointer to the token in **token*. Also accommodates tokens enclosed within matching single quotes, which may contain embedded spaces and escaped single-quote characters. If no token is found, **token* contains NULL on return from this function.

void *acquireSystemMemory(size_t size)

Uses *memalign()* to allocate a block of system memory of length *size*, starting at an address that is guaranteed to be an integral multiple of the size of a pointer to void, and initializes the entire block to binary zeroes. Returns the starting address of the allocated block on success; returns NULL on any error.

int createFile(const char *name, int flags)

Creates a file of the indicated name, using the indicated file creation flags. This function provides common file creation functionality across VxWorks and Unix platforms, invoking *creat()* under VxWorks and *open()* elsewhere. For return values, see *creat*(2) and *open*(2).

unsigned int getInternetAddress(char *hostName)

Returns the IP address of the indicated host machine, or zero if the address cannot be determined.

char *getInternetHostName(unsigned int hostNbr, char *buffer)
Writes the host name of the indicated host machine into *buffer* and returns *buffer*, or returns NULL on any error. The size of *buffer* should be (MAXHOSTNAMELEN + 1).

int getNameOfHost(char *buffer, int bufferLength)
Writes the first (*bufferLength* – 1) characters of the host name of the local machine into *buffer*. Returns 0 on success, –1 on any error.

unsigned int *getAddressOfHost()*
Returns the IP address for the host name of the local machine, or 0 on any error.

void parseSocketSpec(char *socketSpec, unsigned short *portNbr, unsigned int *hostNbr)
Parses *socketSpec*, extracting host number (IP address) and port number from the string. *socketSpec* is expected to be of the form ''{ @ | hostname }[:<portnbr>]'', where @ signifies ''the host name of the local machine''. If host number can be determined, writes it into **hostNbr*; otherwise writes 0 into **hostNbr*. If port number is supplied and is in the range 1024 to 65535, writes it into **portNbr*; otherwise writes 0 into **portNbr*.

void printDottedString(unsigned int hostNbr, char *buffer)
Composes a dotted-string (xxx.xxx.xxx.xxx) representation of the IPv4 address in *hostNbr* and writes that string into *buffer*. The length of *buffer* must be at least 16.

char *getNameOfUser(char *buffer)
Writes the user name of the invoking task or process into *buffer* and returns *buffer*. The size of *buffer* must be at least *L_cuserid*, a constant defined in the stdio.h header file. Returns *buffer*.

int reUseAddress(int fd)
Makes the address that is bound to the socket identified by *fd* reusable, so that the socket can be closed and immediately reopened and re-bound to the same port number. Returns 0 on success, –1 on any error.

int makeIoNonBlocking(int fd)
Makes I/O on the socket identified by *fd* non-blocking; returns –1 on failure. An attempt to read on a non-blocking socket when no data are pending, or to write on it when its output buffer is full, will not block; it will instead return –1 and cause errno to be set to EWOULDBLOCK.

int watchSocket(int fd)
Turns on the ''linger'' and ''keepalive'' options for the socket identified by *fd*. See *socket*(2) for details. Returns 0 on success, –1 on any failure.

void closeOnExec(int fd)
Ensures that *fd* will NOT be open in any child process *fork()*ed from the invoking process. Has no effect on a VxWorks platform.

EXCEPTION REPORTING

The functions in this section offer platform-independent capabilities for reporting on processing exceptions.

The underlying mechanism for ICI's exception reporting is a pair of functions that record error messages in a privately managed pool of static memory. These functions — *postErrmsg()* and *postSysErrmsg()* — are designed to return very rapidly with no possibility of failing, themselves. Nonetheless they are not safe to call from an interrupt service routing (ISR). Although each merely copies its text to the next available location in the error message memory pool, that pool is protected by a mutex; multiple processes might be queued up to take that mutex, so the total time to execute the function is non-deterministic.

Built on top of *postErrmsg()* and *postSysErrmsg()* are the *putErrmsg()* and *putSysErrmsg()* functions, which may take longer to return. Each one simply calls the corresponding ''post'' function but then calls the *writeErrmsgMemos()* function, which calls *writeMemo()* to print (or otherwise deliver) each message currently posted to the pool and then destroys all of those posted messages, emptying the pool.

Recommended general policy on using the ICI exception reporting functions (which the functions in the ION distribution libraries are supposed to adhere to) is as follows:

In the implementation of any ION library function or any ION task's top-level driver function, any condition that prevents the function from continuing execution toward producing the effect it is designed to produce is considered an "error".

Detection of an error should result in the printing of an error message and, normally, the immediate return of whatever return value is used to indicate the failure of the function in which the error was detected. By convention this value is usually -1, but both zero and NULL are appropriate failure indications under some circumstances such as object creation.

The CHKERR, CHKZERO, CHKNULL, and CHKVOID macros are used to implement this behavior in a standard and lexically terse manner. Use of these macros offers an additional feature: for debugging purposes, they can easily be configured to call sm_Abort() to terminate immediately with a core dump instead of returning a error indication. This option is enabled by setting the compiler parameter CORE_FILE_NEEDED to 1 at compilation time.

In the absence of either any error, the function returns a value that indicates nominal completion. By convention this value is usually zero, but under some circumstances other values (such as pointers or addresses) are appropriate indications of nominal completion. Any additional information produced by the function, such as an indication of "success", is usually returned as the value of a reference argument. [Note, though, that database management functions and the SDR hash table management functions deviate from this rule: most return 0 to indicate nominal completion but functional failure (e.g., duplicate key or object not found) and return 1 to indicate functional success.]

So when returning a value that indicates nominal completion of the function -- even if the result might be interpreted as a failure at a higher level (e.g., an object identified by a given string is not found, through no failure of the search function) -- do NOT invoke putErrmsg().

Use putErrmsg() and putSysErrmsg() only when functions are unable to proceed to nominal completion. Use writeMemo() or writeMemoNote() if you just want to log a message.

Whenever returning a value that indicates an error:

> If the failure is due to the failure of a system call or some other non-ION function, assume that errno has already been set by the function at the lowest layer of the call stack; use putSysErrmsg (or postSysErrmsg if in a hurry) to describe the nature of the activity that failed. The text of the error message should normally start with a capital letter

```
                and should NOT end with a period.

                Otherwise -- i.e., the failure is due to a condition
                that was detected within ION -- use putErrmsg (or
                postErrmg if pressed for time) to describe the nature
                of the failure condition.  This will aid in tracing
                the failure through the function stack in which the
                failure was detected.  The text of the error message
                should normally start with a capital letter and should
                end with a period.

        When a failure in a called function is reported to "driver"
        code in an application program, before continuing or exiting
        use writeErrmsgMemos() to empty the message pool and print a
        simple stack trace identifying the failure.
```

char *system_error_msg( )
: Returns a brief text string describing the current system error, as identified by the current value of errno.

void setLogger(Logger usersLoggerName)
: Sets the user function to be used for writing messages to a user-defined ‘‘log’’ medium. The logger function’s calling sequence must match the following prototype:

```
        void    usersLoggerName(char *msg);
```

The default Logger function simply writes the message to standard output.

void writeMemo(char *msg)
: Writes one log message, using the currently defined message logging function.

void writeMemoNote(char *msg, char *note)
: Writes a log message like *writeMemo()*, accompanied by the user-supplied context-specific text in *note*.

void writeErrMemo(char *msg)
: Writes a log message like *writeMemo()*, accompanied by text describing the current system error.

char *itoa(int value)
: Returns a string representation of the signed integer in *value*, nominally for immediate use as an argument to *putErrmsg()*. [Note that the string is constructed in a static buffer; this function is not thread−safe.]

char *utoa(unsigned int value)
: Returns a string representation of the unsigned integer in *value*, nominally for immediate use as an argument to *putErrmsg()*. [Note that the string is constructed in a static buffer; this function is not thread−safe.]

void postErrmsg(char *text, char *argument)
: Constructs an error message noting the name of the source file containing the line at which this function was called, the line number, the *text* of the message, and — if not NULL — a single textual *argument* that can be used to give more specific information about the nature of the reported failure (such as the value of one of the arguments to the failed function). The error message is appended to the list of messages in a privately managed pool of static memory, ERRMSGS_BUFSIZE bytes in length.

If *text* is NULL or is a string of zero length or begins with a newline character (i.e., **text* == ’\0’ or ’\n’), the function returns immediately and no error message is recorded.

The errmsgs pool is designed to be large enough to contain error messages from all levels of the calling stack at the time that an error is encountered. If the remaining unused space in the pool is less

than the size of the new error message, however, the error message is silently omitted. In this case, provided at least two bytes of unused space remain in the pool, a message comprising a single newline character is appended to the list to indicate that a message was omitted due to excessive length.

void postSysErrmsg(char *text, char *arg)
Like *postErrmsg()* except that the error message constructed by the function additionally contains text describing the current system error. *text* is truncated as necessary to assure that the sum of its length and that of the description of the current system error does not exceed 1021 bytes.

int getErrmsg(char *buffer)
Copies the oldest error message in the message pool into *buffer* and removes that message from the pool, making room for new messages. Returns zero if the message pool cannot be locked for update or there are no more messages in the pool; otherwise returns the length of the message copied into *buffer*. Note that, for safety, the size of *buffer* should be ERRMSGS_BUFSIZE.

Note that a returned error message comprising only a single newline character always signifies an error message that was silently omitted because there wasn't enough space left on the message pool to contain it.

void writeErrmsgMemos( )
Calls *getErrmsg()* repeatedly until the message pool is empty, using *writeMemo()* to log all the messages in the pool. Messages that were omitted due to excessive length are indicated by logged lines of the form "[message omitted due to excessive length]".

void putErrmsg(char *text, char *argument)
The *putErrmsg()* function merely calls *postErrmsg()* and then *writeErrmsgMemos()*.

void putSysErrmsg(char *text, char *arg)
The *putSysErrmsg()* function merely calls *postSysErrmsg()* and then *writeErrmsgMemos()*.

void discardErrmsgs( )
Calls *getErrmsg()* repeatedly until the message pool is empty, discarding all of the messages.

void printStackTrace( )
On Linux machines only, uses *writeMemo()* to print a trace of the process's current execution stack, starting with the lowest level of the stack and proceeding to the *main()* function of the executable.

Note that (a) *printStackTrace()* is **only** implemented for Linux platforms at this time; (b) symbolic names of functions can only be printed if the *−rdynamic* flag was enabled when the executable was linked; (c) only the names of non-static functions will appear in the stack trace.

For more complete information about the state of the executable at the time the stack trace snapshot was taken, use the Linux *addr2line* tool. To do this, cd into a directory in which the executable file resides (such as /opt/bin) and submit an addr2line command as follows:

addr2line –e *name_of_executable stack_frame_address*

where both *name_of_executable* and *stack_frame_address* are taken from one of the lines of the printed stack trace. addr2line will print the source file name and line number for that stack frame.

SELF-DELIMITING NUMERIC VALUES (SDNV)

The functions in this section encode and decode SDNVs, portable variable-length numeric variables that expand to whatever size is necessary to contain the values they contain. SDNVs are used extensively in the BP and LTP libraries.

void encodeSdnv(Sdnv *sdnvBuffer, uvast value)
Determines the number of octets of SDNV text needed to contain the value, places that number in the *length* field of the SDNV buffer, and encodes the value in SDNV format into the first *length* octets of the *text* field of the SDNV buffer.

int decodeSdnv(uvast *value, unsigned char *sdnvText)
Determines the length of the SDNV located at *sdnvText* and returns this number after extracting the SDNV's value from those octets and storing it in *value*. Returns 0 if the encoded number value will not

fit into an unsigned vast integer.

ARITHMETIC ON LARGE INTEGERS (SCALARS)

The functions in this section perform simple arithmetic operations on unsigned Scalar objects — structures encapsulating large positive integers in a machine-independent way. Each Scalar comprises two integers, a count of units [ranging from 0 to (2^30 − 1), i.e., up to 1 gig] and a count of gigs [ranging from 0 to (2^31 −1)]. A Scalar can represent a numeric value up to 2 billion billions, i.e., 2 million trillions.

void loadScalar(Scalar *scalar, signed int value)
Sets the value of *scalar* to the absolute value of *value*.

void increaseScalar(Scalar *scalar, signed int value)
Adds to *scalar* the absolute value of *value*.

void reduceScalar(Scalar *scalar, signed int value)
Adds to *scalar* the absolute value of *value*.

void multiplyScalar(Scalar *scalar, signed int value)
Multiplies *scalar* by the absolute value of *value*.

void divideScalar(Scalar *scalar, signed int value)
Divides *scalar* by the absolute value of *value*.

void copyScalar(Scalar *to, Scalar *from)
Copies the value of *from* into *to*.

void addToScalar(Scalar *scalar, Scalar *increment)
Adds *increment* (a Scalar rather than a C integer) to *scalar*.

void subtractFromScalar(Scalar *scalar, Scalar *decrement)
Subtracts *decrement* (a Scalar rather than a C integer) from *scalar*.

int scalarIsValid(Scalar *scalar)
Returns 1 if the arithmetic performed on *scalar* has not resulted in overflow or underflow.

int scalarToSdnv(Sdnv *sdnv, Scalar *scalar)
If *scalar* points to a valid Scalar, stores the value of *scalar* in *sdnv*; otherwise sets the length of *sdnv* to zero.

int sdnvToScalar(Scalar *scalar, unsigned char *sdnvText)
If *sdnvText* points to a sequence of bytes that, when interpreted as the text of an Sdnv, has a value that can be represented in a 61−bit unsigned binary integer, then this function stores that value in *scalar* and returns the detected Sdnv length. Otherwise returns zero.

Note that Scalars and Sdnvs are both representations of potentially large unsigned integer values. Any Scalar can alternatively be represented as an Sdnv. However, it is possible for a valid Sdnv to be too large to represent in a Scalar.

PRIVATE MUTEXES

The functions in this section provide platform-independent management of mutexes for synchronizing operations of threads or tasks in a common private address space.

int initResourceLock(ResourceLock *lock)
Establishes an inter-thread lock for use in locking some resource. Returns 0 if successful, −1 if not.

void killResourceLock(ResourceLock *lock)
Deletes the resource lock referred to by *lock*.

void lockResource(ResourceLock *lock)
Checks the state of *lock*. If the lock is already owned by a different thread, the call blocks until the other thread relinquishes the lock. If the lock is unowned, it is given to the current thread and the lock count is set to 1. If the lock is already owned by this thread, the lock count is incremented by 1.

void unlockResource(ResourceLock *lock)

If called by the current owner of *lock*, decrements *lock*'s lock count by 1; if zero, relinquishes the lock so it may be taken by other threads. Care must be taken to make sure that one, and only one, *unlockResource()* call is issued for each *lockResource()* call issued on a given resource lock.

SHARED MEMORY IPC DEVICES

The functions in this section provide platform-independent management of IPC mechanisms for synchronizing operations of threads, tasks, or processes that may occupy different address spaces but share access to a common system (nominally, processor) memory.

NOTE that this is distinct from the VxWorks ''VxMP'' capability enabling tasks to share access to bus memory or dual-ported board memory from multiple processors. The ''platform'' system will support IPC devices that utilize this capability at some time in the future, but that support is not yet implemented.

int sm_ipc_init( )

Acquires and initializes shared-memory IPC management resources. Must be called before any other shared-memory IPC function is called. Returns 0 on success, –1 on any failure.

void sm_ipc_stop( )

Releases shared-memory IPC management resources, disabling the shared-memory IPC functions until *sm_ipc_init()* is called again.

int sm_GetUniqueKey( )

Some of the ''sm_'' (shared memory) functions described below associate new communication objects with *key* values that uniquely identify them, so that different processes can access them independently. Key values are typically defined as constants in application code. However, when a new communication object is required for which no specific need was anticipated in the application, the *sm_GetUniqueKey()* function can be invoked to obtain a new, arbitrary key value that is known not to be already in use.

sm_SemId sm_SemCreate(int key, int semType)

Creates a shared-memory semaphore that can be used to synchronize activity among tasks or processes residing in a common system memory but possibly multiple address spaces; returns a reference handle for that semaphore, or SM_SEM_NONE on any failure. If *key* refers to an existing semaphore, returns the handle of that semaphore. If *key* is the constant value SM_NO_KEY, automatically obtains an unused key. On VxWorks platforms, *semType* determines the order in which the semaphore is given to multiple tasks that attempt to take it while it is already taken: if set to SM_SEM_PRIORITY then the semaphore is given to tasks in task priority sequence (i.e., the highest-priority task waiting for it receives it when it is released), while otherwise (SM_SEM_FIFO) the semaphore is given to tasks in the order in which they attempted to take it. On all other platforms, only SM_SEM_FIFO behavior is supported and *semType* is ignored.

int sm_SemTake(sm_SemId semId)

Blocks until the indicated semaphore is no longer taken by any other task or process, then takes it. Return 0 on success, –1 on any error.

void sm_SemGive(sm_SemId semId)

Gives the indicated semaphore, so that another task or process can take it.

void sm_SemEnd(sm_SemId semId)

This function is used to pass a termination signal to whatever task is currently blocked on taking the indicated semaphore, if any. It sets to 1 the ''ended'' flag associated with this semaphore, so that a test for *sm_SemEnded()* will return 1, and it gives the semaphore so that the blocked task will have an opportunity to test that flag.

int sm_SemEnded(sm_SemId semId)

This function returns 1 if the ''ended'' flag associated with the indicated semaphore has been set to 1; returns zero otherwise. When the function returns 1 it also gives the semaphore so that any other tasks that might be pended on the same semaphore are also given an opportunity to test it and discover that it has been ended.

void sm_SemUnend(sm_SemId semId)
: This function is used to reset an ended semaphore, so that a restarted subsystem can reuse that semaphore rather than delete it and allocate a new one.

int sm_SemUnwedge(sm_SemId semId, int timeoutSeconds)
: Used to release semaphores that have been taken but never released, possibly because the tasks or processes that took them crashed before releasing them. Attempts to take the semaphore; if this attempt does not succeed within *timeoutSeconds* seconds (providing time for normal processing to be completed, in the event that the semaphore is legitimately and temporarily locked by some task), the semaphore is assumed to be wedged. In any case, the semaphore is then released. Returns 0 on success, –1 on any error.

void sm_SemDelete(sm_SemId semId)
: Destroys the indicated semaphore.

sm_SemId sm_GetTaskSemaphore(int taskId)
: Returns the ID of the semaphore that is dedicated to the private use of the indicated task, or SM_SEM_NONE on any error.

  This function implements the concept that for each task there can always be one dedicated semaphore, which the task can always use for its own purposes, whose key value may be known a priori because the key of the semaphore is based on the task's ID. The design of the function rests on the assumption that each task's ID, whether a VxWorks task ID or a Unix process ID, maps to a number that is out of the range of all possible key values that are arbitrarily produced by *sm_GetUniqueKey()*. For VxWorks, we assume this to be true because task ID is a pointer to task state in memory which we assume not to exceed 2GB; the unique key counter starts at 2GB. For Unix, we assume this to be true because process ID is an index into a process table whose size is less than 64K; unique keys are formed by shifting process ID left 16 bits and adding the value of an incremented counter which is always greater than zero.

int sm_ShmAttach(int key, int size, char **shmPtr, int *id)
: Attaches to a segment of memory to which tasks or processes residing in a common system memory, but possibly multiple address spaces, all have access.

  This function registers the invoking task or process as a user of the shared memory segment identified by *key*. If *key* is the constant value SM_NO_KEY, automatically sets *key* to some unused key value. If a shared memory segment identified by *key* already exists, then *size* may be zero and the value of **shmPtr* is ignored. Otherwise the size of the shared memory segment must be provided in *size* and a new shared memory segment is created in a manner that is dependent on **shmPtr*: if **shmPtr* is NULL then *size* bytes of shared memory are dynamically acquired, allocated, and assigned to the newly created shared memory segment; otherwise the memory located at *shmPtr* is assumed to have been pre-allocated and is merely assigned to the newly created shared memory segment.

  On success, stores the unique shared memory ID of the segment in **id* for possible future destruction, stores a pointer to the segment's assigned memory in **shmPtr*, and returns 1 (if the segment is newly created) or 0 (otherwise). Returns –1 on any error.

void sm_ShmDetach(char *shmPtr)
: Unregisters the invoking task or process as a user of the shared memory starting at *shmPtr*.

void sm_ShmDestroy(int id)
: Destroys the shared memory segment identified by *id*, releasing any memory that was allocated when the segment was created.

PORTABLE MULTI-TASKING

int sm_TaskIdSelf( )
: Returns the unique identifying number of the invoking task or process.

int sm_TaskExists(int taskId)
: Returns non-zero if a task or process identified by *taskId* is currently running on the local processor, zero otherwise.

void *sm_TaskVar(void *arg)
Posts or retrieves the value of the ‘‘task variable’’ belonging to the invoking task. Each task has access to a single task variable, initialized to NULL, that resides in the task’s private state; this can be convenient for passing task-specific information to a signal handler, for example. If *arg* is non-NULL, then **arg* is posted as the new value of the task’s private task variable. In any case, the value of that task variable is returned.

void sm_TaskSuspend( )
Indefinitely suspends execution of the invoking task or process. Helpful if you want to freeze an application at the point at which an error is detected, then use a debugger to examine its state.

void sm_TaskDelay(int seconds)
Same as *snooze* (3).

void sm_TaskYield( )
Relinquishes CPU temporarily for use by other tasks.

int sm_TaskSpawn(char *name, char *arg1, char *arg2, char *arg3, char *arg4, char *arg5, char *arg6, char *arg7, char *arg8, char *arg9, char *arg10, int priority, int stackSize)
Spawns/forks a new task/process, passing it up to ten command-line arguments. *name* is the name of the function (VxWorks) or executable image (UNIX) to be executed in the new task/process.

For UNIX, *name* must be the name of some executable program in the `$PATH` of the invoking process.

For VxWorks, *name* must be the name of some function named in an application-defined private symbol table (if PRIVATE_SYMTAB is defined) or the system symbol table (otherwise). If PRIVATE_SYMTAB is defined, the application must provide a suitable adaptation of the symtab.c source file, which implements the private symbol table.

‘‘priority’’ and ‘‘stackSize’’ are ignored under UNIX. Under VxWorks, if zero they default to the values in the application-defined private symbol table if provided, or otherwise to ICI_PRIORITY (nominally 100) and 32768 respectively.

Returns the task/process ID of the new task/process on success, or −1 on any error.

void sm_TaskKill(int taskId, int sigNbr)
Sends the indicated signal to the indicated task or process.

void sm_TaskDelete(int taskId)
Terminates the indicated task or process.

void *sm_Abort()*
Terminates the calling task or process. If not called while ION is in flight configuration, a stack trace is printed or a core file is written.

int pseudoshell(char *script)
Parses *script* into a command name and up to 10 arguments, then passes the command name and arguments to *sm_TaskSpawn()* for execution. The *sm_TaskSpawn()* function is invoked with priority and stack size both set to zero, causing default values (possibly from an application-defined private symbol table) to be used. Tokens in *script* are normally whitespace-delimited, but a token that is enclosed in single-quote characters (’) may contain embedded whitespace and may contain escaped single-quote characters (‘‘\’’’). On any parsing failure returns −1; otherwise returns the value returned by *sm_TaskSpawn()*.

USER’S GUIDE

Compiling an application that uses ‘‘platform’’:
Just be sure to ‘‘#include ’’platform.h"" at the top of each source file that includes any platform function calls.

Linking/loading an application that uses ‘‘platform’’:

a. In a Solaris environment, link with these libraries:

```
                -lplatform -socket -nsl -posix4 -c

        b.  In a Linux environment, simply link with platform:

                -lplatform

        c.  In a VxWorks environment, use

                ld 1, 0, "libplatform.o"

            to load platform on the target before loading applications.
```

SEE ALSO

gettimeofday(3C)

NAME

psm – Personal Space Management

SYNOPSIS

```
#include "psm.h"

typedef enum { Okay, Redundant, Refused } PsmMgtOutcome;
typedef unsigned long PsmAddress;
typedef struct psm_str
{
        char            *space;
        int             freeNeeded;
        struct psm_str  *trace;
        int             traceArea[3];
} PsmView, *PsmPartition;

[see description for available functions]
```

DESCRIPTION

PSM is a library of functions that support personal space management, that is, user management of an application-configured memory partition. PSM is designed to be faster and more efficient than malloc/free (for details, see the DETAILED DESCRIPTION below), but more importantly it provides a memory management abstraction that insulates applications from differences in the management of private versus shared memory.

PSM is often used to manage shared memory partitions. On most operating systems, separate tasks that connect to a common shared memory partition are given the same base address with which to access the partition. On some systems (such as Solaris) this is not necessarily the case; an absolute address within such a shared partition will be mapped to different pointer values in different tasks. If a pointer value is stored within shared memory and used without conversion by multiple tasks, segment violations will occur.

PSM gets around this problem by providing functions for translating between local pointer values and relative addresses within the shared memory partition. For complete portability, applications which store addresses in shared memory should store these addresses as PSM relative addresses and convert them to local pointer values before using them. The PsmAddress data type is provided for this purpose, along with the conversion functions *psa()* and *psp()*.

int psm_manage(char *start, unsigned int length, char *name, PsmPartition *partitionPointer, PsmMgtOutcome *outcome)

Puts the *length* bytes of memory at *start* under PSM management, associating this memory partition with the identifying string *name* (which is required and which can have a maximum string length of 31). PSM can manage any contiguous range of addresses to which the application has access, typically a block of heap memory returned by a malloc call.

Every other PSM API function must be passed a pointer to a local ‘‘partition’’ state structure characterizing the PSM-managed memory to which the function is to be applied. The partition state structure itself may be pre-allocated in static or local (or shared) memory by the application, in which case a pointer to that structure must be passed to *psm_manage()* as the value of **partitionPointer*; if **partitionPointer* is null, *psm_manage()* will use *malloc()* to allocate this structure dynamically from local memory and will store a pointer to the structure in **partitionPointer*.

psm_manage() formats the managed memory as necessary and returns −1 on any error, 0 otherwise. The outcome to the attempt to manage memory is placed in *outcome*. An outcome of Redundant means that the memory at *start* is already under PSM management with the same name and size. An outcome of Refused means that PSM was unable to put the memory at *start* under PSM management as directed; a diagnostic message was posted to the message pool (see discussion of *putErrmsg()* in *platform* (3)).

char *psm_name(PsmPartition partition)
Returns the name associated with the partition at the time it was put under management.

char *psm_space(PsmPartition partition)
Returns the address of the space managed by PSM for *partition*. This function is provided to enable the application to do an operating-system release (such as *free()*) of this memory when the managed partition is no longer needed. *NOTE* that calling *psm_erase()* or *psm_unmanage()* [or any other PSM function, for that matter] after releasing that space is virtually guaranteed to result in a segmentation fault or other seriously bad behavior.

void *psp(PsmPartition partition, PsmAddress address)
address is an offset within the space managed for the partition. Returns the conversion of that offset into a locally usable pointer.

PsmAddress psa(PsmPartition partition, void *pointer)
Returns the conversion of *pointer* into an offset within the space managed for the partition.

PsmAddress psm_malloc(PsmPartition partition, unsigned int length)
Allocates a block of memory from the ''large pool'' of the indicated partition. (See the DETAILED DESCRIPTION below.) *length* is the size of the block to allocate; the maximum size is 1/2 of the total address space (i.e., 2G for a 32−bit machine). Returns NULL if no free block could be found. The block returned is aligned on a doubleword boundary.

void psm_panic(PsmPartition partition)
Forces the ''large pool'' memory allocation algorithm to hunt laboriously for free blocks in buckets that may not contain any. This setting remains in force for the indicated partition until a subsequent *psm_relax()* call reverses it.

void psm_relax(PsmPartition partition)
Reverses *psm_panic()*. Lets the ''large pool'' memory allocation algorithm return NULL when no free block can be found easily.

PsmAddress psm_zalloc(PsmPartition partition, unsigned int length)
Allocates a block of memory from the ''small pool'' of the indicated partition, if possible; if the requested block size — *length* — is too large for small pool allocation (which is limited to 64 words, i.e., 256 bytes for a 32−bit machine), or if no small pool space is available and the size of the small pool cannot be increased, then allocates from the large pool instead. Small pool allocation is performed by an especially speedy algorithm, and minimum space is consumed in memory management overhead for small-pool blocks. Returns NULL if no free block could be found. The block returned is aligned on a word boundary.

void psm_free(PsmPartition partition, PsmAddress block)
Frees for subsequent re-allocation the indicated block of memory from the indicated partition. *block* may have been allocated by either *psm_malloc()* or *psm_zalloc()*.

int psm_set_root(PsmPartition partition, PsmAddress root)
Sets the ''root'' word of the indicated partition (a word at a fixed, private location in the PSM bookkeeping data area) to the indicated value. This function is typically useful in a shared-memory environment, such as a VxWorks address space, in which a task wants to retrieve from the indicated partition some data that was inserted into the partition by some other task; the partition root word enables multiple tasks to navigate the same data in the same PSM partition in shared memory. The argument is normally a pointer to something like a linked list of the linked lists that populate the partition; in particular, it is likely to be an object catalog (see *psm_add_catlg()*). Returns 0 on success, −1 on any failure (e.g., the partition already has a root object, in which case *psm_erase_root()* must be called before *psm_set_root()*).

PsmAddress psm_get_root(PsmPartition partition)
Retrieves the current value of the root word of the indicated partition.

void psm_erase_root(PsmPartition partition)
Erases the current value of the root word of the indicated partition.

PsmAddress psm_add_catlg(PsmPartition partition)
Allocates space for an object catalog in the indicated partition and establishes the new catalog as the partition's root object. Returns 0 on success, –1 on any error (e.g., the partition already has some other root object).

int psm_catlg(PsmPartition partition, char *objName, PsmAddress objLocation)
Inserts an entry for the indicated object into the catalog that is the root object for this partition. The length of *objName* cannot exceed 32 bytes, and *objName* must be unique in the catalog. Returns 0 on success, –1 on any error.

int psm_uncatlg(PsmPartition partition, char *objName)
Removes the entry for the named object from the catalog that is the root object for this partition, if that object is found in the catalog. Returns 0 on success, –1 on any error.

int psm_locate(PsmPartition partition, char *objName, PsmAddress *objLocation, PsmAddress *entryElt)
Places in **objLocation* the address associated with *objName* in the catalog that is the root object for this partition and places in **entryElt* the address of the list element that points to this catalog entry. If *name* is not found in catalog, set **entryElt* to zero. Returns 0 on success, –1 on any error.

void psm_usage(PsmPartition partition, PsmUsageSummary *summary)
Loads the indicated PsmUsageSummary structure with a snapshot of the indicated partition's usage status. PsmUsageSummary is defined by:

```
typedef struct {
    char            partitionName[32];
    unsigned int    partitionSize;
    unsigned int    smallPoolSize;
    unsigned int    smallPoolFreeBlockCount[SMALL_SIZES];
    unsigned int    smallPoolFree;
    unsigned int    smallPoolAllocated;
    unsigned int    largePoolSize;
    unsigned int    largePoolFreeBlockCount[LARGE_ORDERS];
    unsigned int    largePoolFree;
    unsigned int    largePoolAllocated;
    unsigned int    unusedSize;
} PsmUsageSummary;
```

void psm_report(PsmUsageSummary *summary)
Sends to stdout the content of *summary*, a snapshot of a partition's usage status.

void psm_unmanage(PsmPartition partition)
Terminates local PSM management of the memory in *partition* and destroys the partition state structure **partition*, but doesn't erase anything in the managed memory; PSM management can be re-established by a subsequent call to *psm_manage()*.

void psm_erase(PsmPartition partition)
Unmanages the indicated partition and additionally discards all information in the managed memory, preventing re-management of the partition.

MEMORY USAGE TRACING

If PSM_TRACE is defined at the time the PSM source code is compiled, the system includes built-in support for simple tracing of memory usage: memory allocations are logged, and memory deallocations are matched to logged allocations, ''closing'' them. This enables memory leaks and some other kinds of memory access problems to be readily investigated.

int psm_start_trace(PsmPartition partition, int traceLogSize, char *traceLogAddress)
Begins an episode of PSM memory usage tracing. *traceLogSize* is the number of bytes of shared memory to use for trace activity logging; the frequency with which ''closed'' trace log events must be

deleted will vary inversely with the amount of memory allocated for the trace log. *traceLogAddress* is normally NULL, causing the trace system to allocate *traceLogSize* bytes of shared memory dynamically for trace logging; if non-NULL, it must point to *traceLogSize* bytes of shared memory that have been pre-allocated by the application for this purpose. Returns 0 on success, −1 on any failure.

void psm_print_trace(PsmPartition partition, int verbose)
Prints a cumulative trace report and current usage report for *partition*. If *verbose* is zero, only exceptions (notably, trace log events that remain open — potential memory leaks) are printed; otherwise all activity in the trace log is printed.

void psm_clear_trace(PsmPartition partition)
Deletes all closed trace log events from the log, freeing up memory for additional tracing.

void psm_stop_trace(PsmPartition partition)
Ends the current episode of PSM memory usage tracing. If the shared memory used for the trace log was allocated by *psm_start_trace()*, releases that shared memory.

EXAMPLE

For an example of the use of psm, see the file psmshell.c in the PSM source directory.

USER'S GUIDE

Compiling a PSM application
Just be sure to ''#include ''psm.h"" at the top of each source file that includes any PSM function calls.

Linking/loading a PSM application
a. In a UNIX environment, link with libpsm.a.

b. In a VxWorks environment, use

```
ld 1, 0, "libpsm.o"
```

to load PSM on the target before loading any PSM applications.

Typical usage:
a. Call *psm_manage()* to initiate management of the partition.

b. Call *psm_malloc()* (and/or *psm_zalloc()*) to allocate space in the partition; call *psm_free()* to release space for later re-allocation.

c. When *psm_malloc()* returns NULL and you're willing to wait a while for a more exhaustive free block search, call *psm_panic()* before retrying *psm_malloc()*. When you're no longer so desperate for space, call *psm_relax()*.

d. To store a vital pointer in the single predefined location in the partition that PSM reserves for this purpose, call *psm_set_root()*; to retrieve that pointer, call *psm_get_root()*.

e. To get a snapshot of the current configuration of the partition, call *psm_usage()*. To print this snapshot to stdout, call *psm_report()*.

f. When you're done with the partition but want to leave it in its current state for future re-management (e.g., if the partition is in shared memory), call *psm_unmanage()*. If you're done with the partition forever, call *psm_erase()*.

DETAILED DESCRIPTION

PSM supports user management of an application-configured memory partition. The partition is functionally divided into two pools of variable size: a ''small pool'' of low-overhead blocks aligned on 4−byte boundaries that can each contain up to 256 bytes of user data, and a ''large pool'' of high-overhead blocks aligned on 8−byte boundaries that can each contain up to 2GB of user data.

Space in the small pool is allocated in any one of 64 different block sizes; each possible block size is (4i + n) where i is a ''block list index'' from 1 through 64 and n is the length of the PSM overhead information per block [4 bytes on a 32−bit machine]. Given a user request for a block of size q where q is in the range 1 through 256 inclusive, we return the first block on the j'th small-pool free list where j = (q − 1) / 4. If there

is no such block, we increase the size of the small pool [incrementing its upper limit by (4 * (j + 1)) + n], initialize the increase as a free block from list j, and return that block. No attempt is made to consolidate physically adjacent blocks when they are freed or to bisect large blocks to satisfy requests for small ones; if there is no free block of the requested size and the size of the small pool cannot be increased without encroaching on the large pool (or if the requested size exceeds 256), we attempt to allocate a large-pool block as described below. The differences between small-pool and large-pool blocks are transparent to the user, and small-pool and large-pool blocks can be freely intermixed in an application.

Small-pool blocks are allocated and freed very rapidly, and space overhead consumption is small, but capacity per block is limited and space assigned to small-pool blocks of a given size is never again available for any other purpose. The small pool is designed to satisfy requests for allocation of a stable overall population of small, volatile objects such as List and ListElt structures (see *lyst*(3)).

Space in the large pool is allocated from any one of 29 buckets, one for each power of 2 in the range 8 through 2G. The size of each block can be expressed as (n + 8i + m) where i is any integer in the range 1 through 256M, n is the size of the block's leading overhead area [8 bytes on a 32–bit machine], and m is the size of the block's trailing overhead area [also 8 bytes on a 32–bit machine]. Given a user request for a block of size q where q is in the range 1 through 2G inclusive, we first compute r as the smallest multiple of 8 that is greater than or equal to q. We then allocate the first block in bucket t such that 2 ** (t + 3) is the smallest power of 2 that is greater than r [or, if r is a power of 2, the first block in bucket t such that 2 ** (t + 3) = r]. That is, we try to allocate blocks of size 8 from bucket 0 [2**3 = 8], blocks of size 16 from bucket 1 [2**4 = 16], blocks of size 24 from bucket 2 [2**5 = 32, 32 > 24], blocks of size 32 from bucket 2 [2**5 = 32], and so on. t is the first bucket whose free blocks are ALL guaranteed to be at least as large as r; bucket t − 1 may also contain some blocks that are as large as r (e.g., bucket 1 will contain blocks of size 24 as well as blocks of size 16), but we would have to do a possibly time consuming sequential search through the free blocks in that bucket to find a match, because free blocks within a bucket are stored in no particular order.

If bucket t is empty, we allocate the first block from the first non-empty bucket corresponding to a greater power of two; if all eligible bucket are empty, we increase the size of the large pool [decrementing its lower limit by (r + 16)], initialize the increase as a free block and "free" it, and try again. If the size of the large pool cannot be increased without encroaching on the small pool, then if we are desperate we search sequentially through all blocks in bucket t − 1 (some of which may be of size r or greater) and allocate the first block that is big enough, if any. Otherwise, no block is returned.

Having selected a free block to allocate, we remove the allocated block from the free list, split off as a new free block all bytes in excess of (r + 16) bytes [unless that excess is too small to form a legal-size block], and return the remainder to the user. When a block is freed, it is automatically consolidated with the physically preceding block (if that block is free) and the physically subsequent block (if that block is free).

Large-pool blocks are allocated and freed quite rapidly; capacity is effectively unlimited; space overhead consumption is very high for extremely small objects but becomes an insignificant fraction of block size as block size increases. The large pool is designed to serve as a general-purpose heap with minimal fragmentation whose overhead is best justified when used to store relatively large, long-lived objects such as image packets.

The general goal of this memory allocation scheme is to satisfy memory management requests rapidly and yet minimize the chance of refusing a memory allocation request when adequate unused space exists but is inaccessible (because it is fragmentary or is buried as unused space in a block that is larger than necessary). The size of a small-pool block delivered to satisfy a request for q bytes will never exceed q + 3 (alignment), plus 4 bytes of overhead. The size of a large-pool block delivered to satisfy a request for q bytes will never exceed q + 7 (alignment) + 20 (the maximum excess that can't be split off as a separate free block), plus 16 bytes of overhead.

Neither the small pool nor the large pool ever decrease in size, but large-pool space previously allocated and freed is available for small-pool allocation requests if no small-pool space is available. Small-pool space previously allocated and freed cannot easily be reassigned to the large pool, though, because blocks in the large pool must be physically contiguous to support defragmentation. No such reassignment algorithm has yet been developed.

SEE ALSO

lyst (3)

NAME

sdr – Simple Data Recorder library

SYNOPSIS

```
#include "sdr.h"

[see below for available functions]
```

DESCRIPTION

SDR is a library of functions that support the use of an abstract data recording device called an ‘‘SDR’’ (‘‘simple data recorder’’) for persistent storage of data. The SDR abstraction insulates software not only from the specific characteristics of any single data storage device but also from some kinds of persistent data storage and retrieval chores. The underlying principle is that an SDR provides standardized support for user data organization at object granularity, with direct access to persistent user data objects, rather than supporting user data organization only at ‘‘file’’ granularity and requiring the user to implement access to the data objects accreted within those files.

The SDR library is designed to provide some of the same kinds of directory services as a file system together with support for complex data structures that provide more operational flexibility than files. (As an example of this flexibility, consider how much easier and faster it is to delete a given element from the middle of a linked list than it is to delete a range of bytes from the middle of a text file.) The intent is to enable the software developer to take maximum advantage of the high speed and direct byte addressability of a non-volatile flat address space in the management of persistent data. The SDR equivalent of a ‘‘record’’ of data is simply a block of nominally persistent memory allocated from this address space. The SDR equivalent of a ‘‘file’’ is a *collection* object. Like files, collections can have names, can be located by name within persistent storage, and can impose structure on the data items they encompass. But, as discussed later, SDR collection objects can impose structures other than the strict FIFO accretion of records or bytes that characterizes a file.

The notional data recorder managed by the SDR library takes the form of a single array of randomly accessible, contiguous, nominally persistent memory locations called a *heap*. Physically, the heap may be implemented as a region of shared memory, as a single file of predefined size, or both — that is, the heap may be a region of shared memory that is automatically mirrored in a file.

SDR services that manage SDR data are provided in several layers, each of which relies on the services implemented at lower levels:

At the highest level, a cataloguing service enables retrieval of persistent objects by name.

Services that manage three types of persistent data collections are provided for use both by applications and by the cataloguing service: linked lists, self-delimiting tables (which function as arrays that remember their own dimensions), and self-delimiting strings (short character arrays that remember their lengths, for speedier retrieval).

Basic SDR heap space management services, analogous to *malloc()* and *free()*, enable the creation and destruction of objects of arbitrary type.

Farther down the service stack are memcpy-like low-level functions for reading from and writing to the heap.

Protection of SDR data integrity across a series of reads and writes is provided by a *transaction* mechanism.

SDR persistent data are referenced in application code by Object values and Address values, both of which are simply displacements (offsets) within SDR address space. The difference between the two is that an Object is always the address of a block of heap space returned by some call to *sdr_malloc()*, while an Address can refer to any byte in the address space. That is, an Address is the SDR functional equivalent of a C pointer in DRAM, and some Addresses point to Objects.

Before using SDR services, the services must be loaded to the target machine and initialized by invoking the *sdr_initialize()* function and the management profiles of one or more SDR’s must be loaded by invoking the

sdr_load_profile() function. These steps are normally performed only once, at application load time.

An application gains access to an SDR by passing the name of the SDR to the *sdr_start_using()* function, which returns an Sdr pointer. Most other SDR library functions take an Sdr pointer as first argument.

All writing to an SDR heap must occur during a *transaction* that was initiated by the task issuing the write. Transactions are single-threaded; if task B wants to start a transaction while a transaction begun by task A is still in progress, it must wait until A's transaction is either ended or cancelled. A transaction is begun by calling *sdr_begin_xn()*. The current transaction is normally ended by calling the *sdr_end_xn()* function, which returns an error return code value in the event that any serious SDR-related processing error was encountered in the course of the transaction. Transactions may safely be nested, provided that every level of transaction activity that is begun is properly ended.

The current transaction may instead be cancelled by calling *sdr_cancel_xn()*, which is normally used to indicate that some sort of serious SDR-related processing error has been encountered. Canceling a transaction reverses all SDR update activity performed up to that point within the scope of the transaction — and, if the canceled transaction is an inner, nested transaction, all SDR update activity performed within the scope of every outer transaction encompassing that transaction *and* every other transaction nested within any of those outer transactions — provided the SDR was configured for transaction *reversibility*. When an SDR is configured for reversibility, all heap write operations performed during a transaction are recorded in a log file that is retained until the end of the transaction. Each log file entry notes the location at which the write operation was performed, the length of data written, and the content of the overwritten heap bytes prior to the write operation. Canceling the transaction causes the log entries to be read and processed in reverse order, restoring all overwritten data. Ending the transaction, on the other hand, simply causes the log to be discarded.

If a log file exists at the time that the profile for an SDR is loaded (typically during application initialization), the transaction that was being logged is automatically canceled and reversed. This ensures that, for example, a power failure that occurs in the middle of a transaction will never wreck the SDR's data integrity: either all updates issued during a given transaction are reflected in the current database content or none are.

As a further measure to protect SDR data integrity, an SDR may additionally be configured for *object bounding*. When an SDR is configured to be ''bounded'', every heap write operation is restricted to the extent of a single object allocated from heap space; that is, it's impossible to overwrite part of one object by writing beyond the end of another. To enable the library to enforce this mechanism, application code is prohibited from writing anywhere but within the extent of an object that either (a) was allocated from managed heap space during the same transaction (directly or indirectly via some collection management function) or (b) was *staged* — identified as an update target — during the same transaction (again, either directly or via some collection management function).

Note that both transaction reversibility and object bounding consume processing cycles and inhibit performance to some degree. Determining the right balance between operational safety and processing speed is left to the user.

Note also that, since SDR transactions are single-threaded, they can additionally be used as a general mechanism for simply implementing ''critical sections'' in software that is already using SDR for other purposes: the beginning of a transaction marks the start of code that can't be executed concurrently by multiple tasks. To support this use of the SDR transaction mechanism, the additional transaction termination function *sdr_exit_xn()* is provided. *sdr_exit_xn()* simply ends a transaction without either signaling an error or checking for errors. Like *sdr_cancel_xn()*, *sdr_exit_xn()* has no return value; unlike *sdr_cancel_xn()*, it assures that ending an inner, nested transaction does not cause the outer transaction to be aborted and backed out. But this capability must be used carefully: the protection of SDR data integrity requires that transactions which are ended by *sdr_exit_xn()* must not encompass any SDR update activity whatsoever.

The heap space management functions of the SDR library are adapted directly from the Personal Space Management (*psm*) function library. The manual page for *psm*(3) explains the algorithms used and the rationale behind them. The principal difference between PSM memory management and SDR heap

management is that, for performance reasons, SDR reserves the ''small pool'' for its own use only; all user data space is allocated from the ''large pool'', via the *sdr_malloc()* function.

RETURN VALUES AND ERROR HANDLING

Whenever an SDR function call fails, a diagnostic message explaining the failure of the function is recorded in the error message pool managed by the ''platform'' system (see the discussion of *putErrmsg()* in *platform* (3)).

The failure of any function invoked in the course of an SDR transaction causes all subsequent SDR activity in that transaction to fail immediately. This can streamline SDR application code somewhat: it may not be necessary to check the return value of every SDR function call executed during a transaction. If the *sdr_end_xn()* call returns zero, all updates performed during the transaction must have succeeded.

SYSTEM ADMINISTRATION FUNCTIONS

int sdr_initialize(int wmSize, char *wmPtr, int wmKey, char *wmName)

Initializes the SDR system. *sdr_initialize()* must be called once every time the computer on which the system runs is rebooted, before any call to any other SDR library function.

This function attaches to a pool of shared memory, managed by PSM (see *psm* (3), that enables SDR library operations. If the SDR system is to access a common pool of shared memory with one or more other systems, the key of that shared memory segment must be provided in *wmKey* and the PSM partition name associated with that memory segment must be provided in *wmName*; otherwise *wmKey* must be zero and *wmName* must be NULL, causing *sdr_initialize()* to assign default values. If a shared memory segment identified by the effective value of *wmKey* already exists, then *wmSize* may be zero and the value of *wmPtr* is ignored. Otherwise the size of the shared memory pool must be provided in *wmSize* and a new shared memory segment is created in a manner that is dependent on *wmPtr*: if *wmPtr* is NULL then *wmSize* bytes of shared memory are dynamically acquired, allocated, and assigned to the newly created shared memory segment; otherwise the memory located at *wmPtr* is assumed to have been pre-allocated and is merely assigned to the newly created shared memory segment.

sdr_initialize() also creates a semaphore to serialize access to the SDR system's private array of SDR profiles.

Returns 0 on success, –1 on any failure.

void sdr_wm_usage(PsmUsageSummary *summary)

Loads *summary* with a snapshot of the usage of the SDR system's private working memory. To print the snapshot, use *psm_report()*. (See *psm* (3).)

void sdr_shutdown( )

Ends all access to all SDRs (see *sdr_stop_using()*), detaches from the SDR system's working memory (releasing the memory if it was dynamically allocated by *sdr_initialize()*), and destroys the SDR system's private semaphore. After *sdr_shutdown()*, *sdr_initialize()* must be called again before any call to any other SDR library function.

DATABASE ADMINISTRATION FUNCTIONS

int sdr_load_profile(char *name, int configFlags, long heapWords, int memKey, char *pathName, char *restartCmd, unsigned int restartLatency)

Loads the profile for an SDR into the system's private list of SDR profiles. Although SDRs themselves are persistent, SDR profiles are not: in order for an application to access an SDR, *sdr_load_profile()* must have been called to load the profile of the SDR since the last invocation of *sdr_initialize()*.

name is the name of the SDR, required for any subsequent *sdr_start_using()* call.

configFlags specifies the configuration of the SDR, the bitwise ''or'' of some combination of the following:

SDR_IN_DRAM

SDR is implemented as a region of shared memory.

SDR_IN_FILE
SDR is implemented as a file.

SDR_REVERSIBLE
SDR transactions are logged and are reversed if canceled.

SDR_BOUNDED
Heap updates are not allowed to cross object boundaries.

heapWords specifies the size of the heap in words; word size depends on machine architecture, i.e., a word is 4 bytes on a 32–bit machine, 8 bytes on a 64–bit machine. Note that each SDR prepends to the heap a ''map'' of predefined, fixed size. The total amount of space occupied by an SDR in memory and/or in a file is the sum of the size of the map plus the product of word size and *heapWords*.

memKey is ignored if *configFlags* does not include SDR_IN_DRAM. It should normally be SM_NO_KEY, causing the shared memory region for the SDR to be allocated dynamically and shared using a dynamically selected shared memory key. If specified, *memKey* must be a shared memory key identifying a pre-allocated region of shared memory whose length is equal to the total SDR size, shared via the indicated key.

pathName is ignored if *configFlags* includes neither SDR_REVERSIBLE nor SDR_IN_FILE. It is the fully qualified name of the directory into which the SDR's log file and/or database file will be written. The name of the log file (if any) will be ''<sdrname>.sdrlog''. The name of the database file (if any) will be ''<sdrname>.sdr''; this file will be automatically created and filled with zeros if it does not exist at the time the SDR's profile is loaded.

If a cleanup task must be run whenever a transaction is reversed, the command to execute this task must be provided in *restartCmd* and the number of seconds to wait for this task to finish before resuming operations must be provided in *restartLatency*. If *restartCmd* is NULL or *restartLatency* is zero then no cleanup task will be run upon transaction reversal.

Returns 0 on success, –1 on any error.

int sdr_reload_profile(char *name, int configFlags, long heapWords, int memKey, char *pathName, char *restartCmd, unsigned int restartLatency)
For use when the state of an SDR is thought to be inconsistent, perhaps due to crash of a program that had a transaction open. Unloads the profile for the SDR, forcing the reversal of any transaction that is currently in progress when the SDR's profile is re-loaded. Then calls *sdr_load_profile()* to re-load the profile for the SDR. Same return values as sdr_load_profile.

Sdr sdr_start_using(char *name)
Locates SDR profile by *name* and returns a handle that can be used for all functions that operate on that SDR. On any failure, returns NULL.

char *sdr_name(Sdr sdr)
Returns the name of the sdr.

long sdr_heap_size(Sdr sdr)
Returns the total size of the SDR heap, in bytes.

void sdr_stop_using(Sdr sdr)
Terminates access to the SDR via this handle. Other users of the SDR are not affected. Frees the Sdr object.

void sdr_abort(Sdr sdr)
Terminates the task. In flight configuration, also terminates all use of the SDR system by all tasks.

void sdr_destroy(Sdr sdr)
Ends all access to this SDR, unloads the SDR's profile, and erases the SDR from memory and file system.

DATABASE TRANSACTION FUNCTIONS

void sdr_begin_xn(Sdr sdr)

Initiates a transaction. Note that transactions are single-threaded; any task that calls *sdr_begin_xn()* is suspended until all previously requested transactions have been ended or canceled.

int sdr_in_xn(Sdr sdr)

Returns 1 if called in the course of a transaction, 0 otherwise.

void sdr_exit_xn(Sdr sdr)

Simply abandons the current transaction, ceasing the calling task's lock on ION. Must **not** be used if any database modifications were performed during the transaction; *sdr_end_xn()* must be called instead, to commit those modifications.

void sdr_cancel_xn(Sdr sdr)

Cancels the current transaction. If reversibility is enabled for the SDR, canceling a transaction reverses all heap modifications performed during that transaction.

int sdr_end_xn(Sdr sdr)

Ends the current transaction. Returns 0 if the transaction completed without any error; returns −1 if any operation performed in the course of the transaction failed, in which case the transaction was automatically canceled.

DATABASE I/O FUNCTIONS

void sdr_read(Sdr sdr, char *into, Address from, int length)

Copies *length* characters at *from* (a location in the indicated SDR) to the memory location given by *into*. The data are copied from the shared memory region in which the SDR resides, if any; otherwise they are read from the file in which the SDR resides.

void sdr_peek(sdr, variable, from)

sdr_peek() is a macro that uses *sdr_read()* to load *variable* from the indicated address in the SDR database; the size of *variable* is used as the number of bytes to copy.

void sdr_write(Sdr sdr, Address into, char *from, int length)

Copies *length* characters at *from* (a location in memory) to the SDR heap location given by *into*. Can only be performed during a transaction, and if the SDR is configured for object bounding then heap locations *into* through (*into* + (*length* − 1)) must be within the extent of some object that was either allocated or staged within the same transaction. The data are copied both to the shared memory region in which the SDR resides, if any, and also to the file in which the SDR resides, if any.

void sdr_poke(sdr, into, variable)

sdr_poke() is a macro that uses *sdr_write()* to store *variable* at the indicated address in the SDR database; the size of *variable* is used as the number of bytes to copy.

char *sdr_pointer(Sdr sdr, Address address)

Returns a pointer to the indicated location in the heap – a "heap pointer" – or NULL if the indicated address is invalid. NOTE that this function *cannot be used* if the SDR does not reside in a shared memory region.

Providing an alternative to using *sdr_read()* to retrieve objects into local memory, *sdr_pointer()* can help make SDR-based applications run very quickly, but it must be used WITH GREAT CAUTION! Never use a direct pointer into the heap when not within a transaction, because you will have no assurance at any time that the object pointed to by that pointer has not changed (or is even still there). And NEVER de-reference a heap pointer in order to write directly into the heap: this makes transaction reversal impossible. Whenever writing to the SDR, always use *sdr_write()*.

Address sdr_address(Sdr sdr, char *pointer)

Returns the address within the SDR heap of the indicated location, which must be (or be derived from) a heap pointer as returned by *sdr_pointer()*. Returns zero if the indicated location is not greater than the start of the heap mirror. NOTE that this function *cannot be used* if the SDR does not reside in a shared memory region.

void sdr_get(sdr, variable, heap_pointer)
sdr_get() is a macro that uses *sdr_read()* to load *variable* from the SDR address given by *heap_pointer*; *heap_pointer* must be (or be derived from) a heap pointer as returned by *sdr_pointer()*. The size of *variable* is used as the number of bytes to copy.

void sdr_set(sdr, heap_pointer, variable)
sdr_set() is a macro that uses *sdr_write()* to store *variable* at the SDR address given by *heap_pointer*; *heap_pointer* must be (or be derived from) a heap pointer as returned by *sdr_pointer()*. The size of *variable* is used as the number of bytes to copy.

HEAP SPACE MANAGEMENT FUNCTIONS

Object sdr_malloc(Sdr sdr, unsigned long size)
Allocates a block of space from the of the indicated SDR's heap. *size* is the size of the block to allocate; the maximum size is 1/2 of the maximum address space size (i.e., 2G for a 32−bit machine). Returns block address if successful, zero if block could not be allocated.

Object sdr_insert(Sdr sdr, char *from, unsigned long size)
Uses *sdr_malloc()* to obtain a block of space of size *size* and, if this allocation is successful, uses *sdr_write()* to copy *size* bytes of data from memory at *from* into the newly allocated block. Returns block address if successful, zero if block could not be allocated.

Object sdr_stow(sdr, variable)
sdr_stow() is a macro that uses *sdr_insert()* to insert a copy of *variable* into the database. The size of *variable* is used as the number of bytes to copy.

int sdr_object_length(Sdr sdr, Object object)
Returns the number of bytes of heap space allocated to the application data at *object*.

void sdr_free(Sdr sdr, Object object)
Frees for subsequent re-allocation the heap space occupied by *object*.

void sdr_stage(Sdr sdr, char *into, Object from, int length)
Like *sdr_read()*, this function will copy *length* characters at *from* (a location in the heap of the indicated SDR) to the memory location given by *into*. Unlike *sdr_get()*, *sdr_stage()* requires that *from* be the address of some allocated object, not just any location within the heap. *sdr_stage()*, when called from within a transaction, notifies the SDR library that the indicated object may be updated later in the transaction; this enables the library to retrieve the object's size for later reference in validating attempts to write into some location within the object. If *length* is zero, the object's size is privately retrieved by SDR but none of the object's content is copied into memory.

long sdr_unused(Sdr sdr)
Returns number of bytes of heap space not yet allocated to either the large or small objects pool.

void sdr_usage(Sdr sdr, SdrUsageSummary *summary)
Loads the indicated SdrUsageSummary structure with a snapshot of the SDR's usage status. SdrUsageSummary is defined by:

```
typedef struct
{
        char            sdrName[MAX_SDR_NAME + 1];
        unsigned int    sdrSize;
        unsigned int    smallPoolSize;
        unsigned int    smallPoolFreeBlockCount[SMALL_SIZES];
        unsigned int    smallPoolFree;
        unsigned int    smallPoolAllocated;
        unsigned int    largePoolSize;
        unsigned int    largePoolFreeBlockCount[LARGE_ORDERS];
        unsigned int    largePoolFree;
        unsigned int    largePoolAllocated;
        unsigned int    unusedSize;
```

```
} SdrUsageSummary;
```

void sdr_report(SdrUsageSummary *summary)
Sends to stdout a printed summary of the SDR's usage status.

int sdr_heap_depleted(Sdr sdr)
A Boolean function: returns 1 if the total available space in the SDR's heap (small pool free, large pool free, and unused) is less than 1/16 of the total size of the heap. Otherwise returns zero.

HEAP SPACE USAGE TRACING

If SDR_TRACE is defined at the time the SDR source code is compiled, the system includes built-in support for simple tracing of SDR heap space usage: heap space allocations are logged, and heap space deallocations are matched to logged allocations, ''closing'' them. This enables heap space leaks and some other kinds of SDR heap access problems to be readily investigated.

int sdr_start_trace(Sdr sdr, int traceLogSize, char *traceLogAddress)
Begins an episode of SDR heap space usage tracing. *traceLogSize* is the number of bytes of shared memory to use for trace activity logging; the frequency with which ''closed'' trace log events must be deleted will vary inversely with the amount of memory allocated for the trace log. *traceLogAddress* is normally NULL, causing the trace system to allocate *traceLogSize* bytes of shared memory dynamically for trace logging; if non-NULL, it must point to *traceLogSize* bytes of shared memory that have been pre-allocated by the application for this purpose. Returns 0 on success, –1 on any failure.

void sdr_print_trace(Sdr sdr, int verbose)
Prints a cumulative trace report and current usage report for *sdr*. If *verbose* is zero, only exceptions (notably, trace log events that remain open — potential SDR heap space leaks) are printed; otherwise all activity in the trace log is printed.

void sdr_clear_trace(Sdr sdr)
Deletes all closed trace log events from the log, freeing up memory for additional tracing.

void sdr_stop_trace(Sdr sdr)
Ends the current episode of SDR heap space usage tracing. If the shared memory used for the trace log was allocated by *sdr_start_trace()*, releases that shared memory.

CATALOGUE FUNCTIONS

The SDR catalogue functions are used to maintain the catalogue of the names, types, and addresses of objects within an SDR. The catalogue service includes functions for creating, deleting and finding catalogue entries and a function for navigating through catalogue entries sequentially.

void sdr_catlg(Sdr sdr, char *name, int type, Object object)
Associates *object* with *name* in the indicated SDR's catalogue and notes the *type* that was declared for this object. *type* is optional and has no significance other than that conferred on it by the application.

The SDR catalogue is flat, not hierarchical like a directory tree, and all names must be unique. The length of *name* is limited to 15 characters.

Object sdr_find(Sdr sdr, char *name, int *type)
Locates the Object associated with *name* in the indicated SDR's catalogue and returns its address; also reports the catalogued type of the object in **type* if *type* is non-NULL. Returns zero if no object is currently catalogued under this name.

void sdr_uncatlg(Sdr sdr, char *name)
Dissociates from *name* whatever object in the indicated SDR's catalogue is currently catalogued under that name.

Object sdr_read_catlg(Sdr sdr, char *name, int *type, Object *object, Object previous_entry)
Used to navigate through catalogue entries sequentially. If *previous_entry* is zero, reads the first entry in the indicated SDR's catalogue; otherwise, reads the next catalogue entry following the one located at *previous_entry*. In either case, returns zero if no such catalogue entry exists; otherwise, copies that entry's name, type, and catalogued object address into *name*, **type*, and **object*, and then returns the

address of the catalogue entry (which may be used as *previous_entry* in a subsequent call to *sdr_read_catlg()*).

USER'S GUIDE

Compiling an SDR application

Just be sure to ''#include ''sdr.h"" at the top of each source file that includes any SDR function calls.

For UNIX applications, link with ''–lsdr''.

Loading an SDR application (VxWorks)

```
ld < "libsdr.o"
```

After the library has been loaded, you can begin loading SDR applications.

SEE ALSO

sdrlist (3), *sdrstring* (3), *sdrtable* (3)

NAME

sdrhash – Simple Data Recorder hash table management functions

SYNOPSIS

```
#include "sdr.h"

Object  sdr_hash_create        (Sdr sdr, int keyLength,
                                int estNbrOfEntries,
                                int meanSearchLength);
int     sdr_hash_insert        (Sdr sdr, Object hash, char *key,
                                Address value, Object *entry);
int     sdr_hash_delete_entry  (Sdr sdr, Object entry);
int     sdr_hash_entry_value   (Sdr sdr, Object hash, Object entry);
int     sdr_hash_retrieve      (Sdr sdr, Object hash, char *key,
                                Address *value, Object *entry);
int     sdr_hash_count         (Sdr sdr, Object hash);
int     sdr_hash_revise        (Sdr sdr, Object hash, char *key,
                                Address value);
int     sdr_hash_remove        (Sdr sdr, Object hash, char *key,
                                Address *value);
int     sdr_hash_destroy       (Sdr sdr, Object hash);
```

DESCRIPTION

The SDR hash functions manage hash table objects in an SDR.

Hash tables associate values with keys. A value is always in the form of an SDR Address, nominally the address of some stored object identified by the associated key, but the actual significance of a value may be anything that fits into a *long*. A key is always an array of from 1 to 255 bytes, which may have any semantics at all.

Keys must be unique; no two distinct entries in an SDR hash table may have the same key. Any attempt to insert a duplicate entry in an SDR hash table will be rejected.

All keys must be of the same length, and that length must be declared at the time the hash table is created. Invoking a hash table function with a key that is shorter than the declared length will have unpredictable results.

An SDR hash table is an array of linked lists. The location of a given value in the hash table is automatically determined by computing a ''hash'' of the key, dividing the hash by the number of linked lists in the array, using the remainder as an index to the corresponding linked list, and then sequentially searching through the list entries until the entry with the matching key is found.

The number of linked lists in the array is automatically computed at the time the hash table is created, based on the estimated maximum number of entries you expect to store in the table and the mean linked list length (i.e., mean search time) you prefer. Increasing the maximum number of entries in the table and decreasing the mean linked list length both tend to increase the amount of SDR heap space occupied by the hash table.

Object sdr_hash_create(Sdr sdr, int keyLength, int estNbrOfEntries, int meanSearchLength)

Creates an SDR hash table. Returns the SDR address of the new hash table on success, zero on any error.

int sdr_hash_insert(Sdr sdr, Object hash, char *key, Address value, Object *entry)

Inserts an entry into the hash table identified by *hash*. On success, places the address of the new hash table entry in *entry* and returns zero. Returns –1 on any error.

int sdr_hash_delete_entry(Sdr sdr, Object entry)

Deletes the hash table entry identified by *entry*. Returns zero on success, –1 on any error.

Address sdr_hash_entry_value(Sdr sdr, Object hash, Object entry)
Returns the value of the hash table entry identified by *entry*.

int sdr_hash_retrieve(Sdr sdr, Object hash, char *key, Address *value, Object *entry)
Searches for the value associated with *key* in this hash table, storing it in *value* if found. If the entry matching *key* was found, places the address of the hash table entry in *entry* and returns 1. Returns zero if no such entry exists, –1 on any other failure.

int sdr_hash_count(Sdr sdr, Object hash)
Returns the number of entries in the hash table identified by *hash*.

int sdr_hash_revise(Sdr sdr, Object hash, char *key, Address value)
Searches for the hash table entry matching *key* in this hash table, replacing the associated value with *value* if found. Returns 1 if the entry matching *key* was found, zero if no such entry exists, –1 on any other failure.

int sdr_hash_remove(Sdr sdr, Object hash, char *key, Address *value)
Searches for the hash table entry matching *key* in this hash table; if the entry is found, stores its value in *value*, deletes the entry, and returns 1. Returns zero if no such entry exists, –1 on any other failure.

void sdr_hash_destroy(Sdr sdr, Object hash);
Destroys *hash*, destroying all entries in all linked lists of the array and destroying the hash table array structure itself. DO NOT use *sdr_free()* to destroy a hash table, as this would leave the hash table's content allocated yet unreferenced.

SEE ALSO

sdr (3), *sdrlist* (3), *sdrtable* (3)

NAME

sdrlist – Simple Data Recorder list management functions

SYNOPSIS

```
#include "sdr.h"

typedef int (*SdrListCompareFn)(Sdr sdr, Address eltData, void *argData);
typedef void (*SdrListDeleteFn)(Sdr sdr, Object elt, void *argument);

[see description for available functions]
```

DESCRIPTION

The SDR list management functions manage doubly-linked lists in managed SDR heap space. The functions manage two kinds of objects: lists and list elements. A list knows how many elements it contains and what its start and end elements are. An element knows what list it belongs to and the elements before and after it in the list. An element also knows its content, which is normally the SDR Address of some object in the SDR heap. A list may be sorted, which speeds the process of searching for a particular element.

Object sdr_list_create(Sdr sdr)

Creates a new list object in the SDR; the new list object initially contains no list elements. Returns the address of the new list, or zero on any error.

void sdr_list_destroy(Sdr sdr, Object list, SdrListDeleteFn fn, void *arg)

Destroys a list, freeing all elements of list. If *fn* is non-NULL, that function is called once for each freed element; when called, *fn* is passed the Address that is the element's data and the *argument* pointer passed to *sdr_list_destroy()*.

Do not use *sdr_free* to destroy an SDR list, as this would leave the elements of the list allocated yet unreferenced.

int sdr_list_length(Sdr sdr, Object list)

Returns the number of elements in the list, or –1 on any error.

void sdr_list_user_data_set(Sdr sdr, Object list, Address userData)

Sets the ''user data'' word of *list* to *userData*. Note that *userData* is nominally an Address but can in fact be any value that occupies a single word. It is typically used to point to an SDR object that somehow characterizes the list as a whole, such as a name.

Address sdr_list_user_data(Sdr sdr, Object list)

Returns the value of the ''user data'' word of *list*, or zero on any error.

Object sdr_list_insert(Sdr sdr, Object list, Address data, SdrListCompareFn fn, void *dataBuffer)

Creates a new list element whose data value is *data* and inserts that element into the list. If *fn* is NULL, the new list element is simply appended to the list; otherwise, the new list element is inserted after the last element in the list whose data value is ''less than or equal to'' the data value of the new element (in dataBuffer) according to the collating sequence established by *fn*. Returns the address of the newly created element, or zero on any error.

Object sdr_list_insert_first(Sdr sdr, Object list, Address data)
Object sdr_list_insert_last(Sdr sdr, Object list, Address data)

Creates a new element and inserts it at the front/end of the list. This function should not be used to insert a new element into any ordered list; use *sdr_list_insert()* instead. Returns the address of the newly created list element on success, or zero on any error.

Object sdr_list_insert_before(Sdr sdr, Object elt, Address data)
Object sdr_list_insert_after(Sdr sdr, Object elt, Address data)

Creates a new element and inserts it before/after the specified list element. This function should not be used to insert a new element into any ordered list; use *sdr_list_insert()* instead. Returns the address of the newly created list element, or zero on any error.

void sdr_list_delete(Sdr sdr, Object elt, SdrListDeleteFn fn, void *arg)
Delete *elt* from the list it is in. If *fn* is non-NULL, that function will be called upon deletion of *elt*; when called, that function is passed the Address that is the list element's data value and the *arg* pointer passed to *sdr_list_delete()*.

Object sdr_list_first(Sdr sdr, Object list)
Object sdr_list_last(Sdr sdr, Object list)
Returns the address of the first/last element of *list*, or zero on any error.

Object sdr_list_next(Sdr sdr, Object elt)
Object sdr_list_prev(Sdr sdr, Object elt)
Returns the address of the element following/preceding *elt* in that element's list, or zero on any error.

Object sdr_list_search(Sdr sdr, Object elt, int reverse, SdrListCompareFn fn, void *dataBuffer);
Search a list for an element whose data matches the data in *dataBuffer*, starting at the indicated initial list element. If the *compare* function is non-NULL, the list is assumed to be sorted in the order implied by that function and the function is automatically called once for each element of the list until it returns a value that is greater than or equal to zero (where zero indicates an exact match and a value greater than zero indicates that the list contains no matching element); each time *compare* is called it is passed the Address that is the element's data value and the *dataBuffer* value passed to *sm_list_search()*. If *reverse* is non-zero, then the list is searched in reverse order (starting at the indicated initial list element) and the search ends when *compare* returns a value that is less than or equal to zero. If *compare* is NULL, then the entire list is searched (in either forward or reverse order, as directed) until an element is located whose data value is equal to ((Address) *dataBuffer*). Returns the address of the matching element if one is found, 0 otherwise.

Object sdr_list_list(Sdr sdr, Object elt)
Returns the address of the list to which *elt* belongs, or 0 on any error.

Address sdr_list_data(Sdr sdr, Object elt)
Returns the Address that is the data value of *elt*, or 0 on any error.

Address sdr_list_data_set(Sdr sdr, Object elt, Address data)
Sets the data value for *elt* to *data*, replacing the original value. Returns the original data value for *elt*, or 0 on any error. The original data value for *elt* may or may not have been the address of an object in heap data space; even if it was, that object was NOT deleted.

Warning: changing the data value of an element of an ordered list may ruin the ordering of the list.

USAGE

When inserting elements or searching a list, the user may optionally provide a compare function of the form:

```
int user_comp_name(Sdr sdr, Address eltData, void *dataBuffer);
```

When provided, this function is automatically called by the sdrlist function being invoked; when the function is called it is passed the content of a list element (*eltData*, nominally the Address of an item in the SDR's heap space) and an argument, *dataBuffer*, which is nominally the address in local memory of some other item in the same format. The user-supplied function normally compares some key values of the two data items and returns 0 if they are equal, an integer less than 0 if *eltData*'s key value is less than that of *dataBuffer*, and an integer greater than 0 if *eltData*'s key value is greater than that of *dataBuffer*. These return values will produce a list in ascending order. If the user desires the list to be in descending order, the function must reverse the signs of these return values.

When deleting an element or destroying a list, the user may optionally provide a delete function of the form:

```
void user_delete_name(Sdr sdr, Address eltData, void *argData)
```

When provided, this function is automatically called by the sdrlist function being invoked; when the function is called it is passed the content of a list element (*eltData*, nominally the Address of an item in the SDR's heap space) and an argument, *argData*, which if non-NULL is normally the address in local memory

of a data item providing context for the list element deletion. The user-supplied function performs any application-specific cleanup associated with deleting the element, such as freeing the element's content data item and/or other SDR heap space associated with the element.

SEE ALSO

lyst (3), *sdr* (3), *sdrstring* (3), *sdrtable* (3), *smlist* (3)

NAME

sdrstring – Simple Data Recorder string functions

SYNOPSIS

```
#include "sdr.h"

Object sdr_string_create (Sdr sdr, char *from);
Object sdr_string_dup    (Sdr sdr, Object from);
int    sdr_string_length (Sdr sdr, Object string);
int    sdr_string_read   (Sdr sdr, char *into, Object string);
```

DESCRIPTION

SDR strings are used to record strings of up to 255 ASCII characters in the heap space of an SDR. Unlike standard C strings, which are terminated by a zero byte, SDR strings record the length of the string as part of the string object.

To store strings longer than 255 characters, use *sdr_malloc()* and *sdr_write()* instead of these functions.

Object sdr_string_create(Sdr sdr, char *from)

Creates a ''self-delimited string'' in the heap of the indicated SDR, allocating the required space and copying the indicated content. *from* must be a standard C string for which *strlen()* must not exceed 255; if it does, or if insufficient SDR space is available, 0 is returned. Otherwise the address of the newly created SDR string object is returned. To destroy, just use *sdr_free()*.

Object sdr_string_dup(Sdr sdr, Object from)

Creates a duplicate of the SDR string whose address is *from*, allocating the required space and copying the original string's content. If insufficient SDR space is available, 0 is returned. Otherwise the address of the newly created copy of the original SDR string object is returned. To destroy, use *sdr_free()*.

int sdr_string_length(Sdr sdr, Object string)

Returns the length of the indicated self-delimited string (as would be returned by *strlen()*), or −1 on any error.

int sdr_string_read(Sdr sdr, char *into, Object string)

Retrieves the content of the indicated self-delimited string into memory as a standard C string (NULL terminated). Length of *into* should normally be SDRSTRING_BUFSZ (i.e., 256) to allow for the largest possible SDR string (255 characters) plus the terminating NULL. Returns length of string (as would be returned by *strlen()*), or −1 on any error.

SEE ALSO

sdr (3), *sdrlist* (3), *sdrtable* (3), *string* (3)

NAME

sdrtable – Simple Data Recorder table management functions

SYNOPSIS

```
#include "sdr.h"

Object  sdr_table_create        (Sdr sdr, int rowSize, int rowCount);
int     sdr_table_user_data_set (Sdr sdr, Object table, Address userData);
Address sdr_table_user_data     (Sdr sdr, Object table);
int     sdr_table_dimensions    (Sdr sdr, Object table, int *rowSize,
                                    int *rowCount);
int     sdr_table_stage         (Sdr sdr, Object table);
Address sdr_table_row           (Sdr sdr, Object table,
                                    unsigned int rowNbr);
int     sdr_table_destroy       (Sdr sdr, Object table);
```

DESCRIPTION

The SDR table functions manage table objects in the SDR. An SDR table comprises N rows of M bytes each, plus optionally one word of user data (which is nominally the address of some other object in the SDR's heap space). When a table is created, the number of rows in the table and the length of each row are specified; they remain fixed for the life of the table. The table functions merely maintain information about the table structure and its location in the SDR and calculate row addresses; other SDR functions such as *sdr_read()* and *sdr_write()* are used to read and write the contents of the table's rows. In particular, the format of the rows of a table is left entirely up to the user.

Object sdr_table_create(Sdr sdr, int rowSize, int rowCount)

Creates a ''self-delimited table'', comprising *rowCount* rows of *rowSize* bytes each, in the heap space of the indicated SDR. Note that the content of the table, a two-dimensional array, is a single SDR heap space object of size (*rowCount* x *rowSize*). Returns the address of the new table on success, zero on any error.

void sdr_table_user_data_set(Sdr sdr, Object table, Address userData)

Sets the ''user data'' word of *table* to *userData*. Note that *userData* is nominally an Address but can in fact be any value that occupies a single word. It is typically used to point to an SDR object that somehow characterizes the table as a whole, such as an SDR string containing a name.

Address sdr_table_user_data(Sdr sdr, Object table)

Returns the value of the ''user data'' word of *table*, or zero on any error.

void sdr_table_dimensions(Sdr sdr, Object table, int *rowSize, int *rowCount)

Reports on the row size and row count of the indicated table, as specified when the table was created.

void sdr_table_stage(Sdr sdr, Object table)

Stages *table* so that the array it encapsulates may be updated; see the discussion of *sdr_stage()* in *sdr* (3). The effect of this function is the same as:

```
sdr_stage(sdr, NULL, (Object) sdr_table_row(sdr, table, 0), 0)
```

Address sdr_table_row(Sdr sdr, Object table, unsigned int rowNbr)

Returns the address of the *rowNbr*th row of *table*, for use in reading or writing the content of this row; returns –1 on any error.

void sdr_table_destroy(Sdr sdr, Object table)

Destroys *table*, releasing all bytes of all rows and destroying the table structure itself. DO NOT use *sdr_free()* to destroy a table, as this would leave the table's content allocated yet unreferenced.

SEE ALSO

sdr (3), *sdrlist* (3), *sdrstring* (3)

NAME

smlist – shared memory list management library

SYNOPSIS

```
#include "smlist.h"

typedef int (*SmListCompareFn)
    (PsmPartition partition, PsmAddress eltData, void *argData);
typedef void (*SmListDeleteFn)
    (PsmPartition partition, PsmAddress eltData, void *argument);

[see description for available functions]
```

DESCRIPTION

The smlist library provides functions to create, manipulate and destroy doubly-linked lists in shared memory. As with *lyst*(3), smlist uses two types of objects, *list* objects and *element* objects. However, as these objects are stored in shared memory which is managed by *psm*(3), pointers to these objects are carried as PsmAddress values. A list knows how many elements it contains and what its first and last elements are. An element knows what list it belongs to and the elements before and after it in its list. An element also knows its content, which is normally the PsmAddress of some object in shared memory.

PsmAddress sm_list_create(PsmPartition partition)

Create a new list object without any elements in it, within the memory segment identified by *partition*. Returns the PsmAddress of the list, or 0 on any error.

void sm_list_unwedge(PsmPartition partition, PsmAddress list, int interval)

Unwedge, as necessary, the mutex semaphore protecting shared access to the indicated list. For details, see the explanation of the *sm_SemUnwedge()* function in *platform*(3).

int sm_list_clear(PsmPartition partition, PsmAddress list, SmListDeleteFn delete, void *argument);

Empty a list. Frees each element of the list. If the *delete* function is non-NULL, that function is called once for each freed element; when called, that function is passed the PsmAddress that is the element's data and the *argument* pointer passed to *sm_list_clear()*. Returns 0 on success, –1 on any error.

int sm_list_destroy(PsmPartition partition, PsmAddress list, SmListDeleteFn delete, void *argument);

Destroy a list. Same as *sm_list_clear()*, but additionally frees the list structure itself. Returns 0 on success, –1 on any error.

int sm_list_user_data_set(PsmPartition partition, PsmAddress list, PsmAddress userData);

Set the value of a user data variable associated with the list as a whole. This value may be used for any purpose; it is typically used to store the PsmAddress of a shared memory block containing data (e.g., state data) which the user wishes to associate with the list. Returns 0 on success, –1 on any error.

PsmAddress sm_list_user_data(PsmPartition partition, PsmAddress list);

Return the value of the user data variable associated with the list as a whole, or 0 on any error.

int sm_list_length(PsmPartition partition, PsmAddress list);

Return the number of elements in the list.

PsmAddress sm_list_insert(PsmPartition partition, PsmAddress list, PsmAddress data, SmListCompareFn compare, void *dataBuffer);

Create a new list element whose data value is *data* and insert it into the given list. If the *compare* function is NULL, the new list element is simply appended to the list; otherwise, the new list element is inserted after the last element in the list whose data value is ''less than or equal to'' the data value of the new element (in *dataBuffer*) according to the collating sequence established by *compare*. Returns the PsmAddress of the new element, or 0 on any error.

PsmAddress sm_list_insert_first(PsmPartition partition, PsmAddress list, PsmAddress data);

PsmAddress sm_list_insert_last(PsmPartition partition, PsmAddress list, PsmAddress data);
Create a new list element and insert it at the start/end of a list. Returns the PsmAddress of the new element on success, or 0 on any error. Disregards any established sort order in the list.

PsmAddress sm_list_insert_before(PsmPartition partition, PsmAddress elt, PsmAddress data);
PsmAddress sm_list_insert_after(PsmPartition partition, PsmAddress elt, PsmAddress data);
Create a new list element and insert it before/after a given element. Returns the PsmAddress of the new element on success, or 0 on any error. Disregards any established sort order in the list.

int sm_list_delete(PsmPartition partition, PsmAddress elt, SmListDeleteFn delete, void *argument);
Delete an element from a list. If the *delete* function is non-NULL, that function is called upon deletion of *elt*; when called, that function is passed the PsmAddress that is the element's data value and the *argument* pointer passed to *sm_list_delete()*. Returns 0 on success, –1 on any error.

PsmAddress sm_list_first(PsmPartition partition, PsmAddress list);
PsmAddress sm_list_last(PsmPartition partition, PsmAddress list);
Return the PsmAddress of the first/last element in *list*, or 0 on any error.

PsmAddress sm_list_next(PsmPartition partition, PsmAddress elt);
PsmAddress sm_list_prev(PsmPartition partition, PsmAddress elt);
Return the PsmAddress of the element following/preceding *elt* in that element's list, or 0 on any error.

PsmAddress sm_list_search(PsmPartition partition, PsmAddress elt, SmListCompareFn compare, void *dataBuffer);
Search a list for an element whose data matches the data in *dataBuffer*. If the *compare* function is non-NULL, the list is assumed to be sorted in the order implied by that function and the function is automatically called once for each element of the list until it returns a value that is greater than or equal to zero (where zero indicates an exact match and a value greater than zero indicates that the list contains no matching element); each time *compare* is called it is passed the PsmAddress that is the element's data value and the *dataBuffer* value passed to *sm_list_search()*. If *compare* is NULL, then the entire list is searched until an element is located whose data value is equal to ((PsmAddress) *dataBuffer*). Returns the PsmAddress of the matching element if one is found, 0 otherwise.

PsmAddress sm_list_list(PsmPartition partition, PsmAddress elt);
Return the PsmAddress of the list to which *elt* belongs, or 0 on any error.

PsmAddress sm_list_data(PsmPartition partition, PsmAddress elt);
Return the PsmAddress that is the data value for *elt*, or 0 on any error.

PsmAddress sm_list_data_set(PsmPartition partition, PsmAddress elt, PsmAddress data);
Set the data value for *elt* to *data*, replacing the original value. Returns the original data value for *elt*, or 0 on any error. The original data value for *elt* may or may not have been the address of an object in memory; even if it was, that object was NOT deleted.

Warning: changing the data value of an element of an ordered list may ruin the ordering of the list.

USAGE

A user normally creates an element and adds it to a list by doing the following:

1. obtaining a shared memory block to contain the element's data;
2. converting the shared memory block's PsmAddress to a character pointer;
3. using that pointer to write the data into the shared memory block;
4. calling one of the *sm_list_insert* functions to create the element structure (which will include the shared memory block's PsmAddress) and insert it into the list.

When inserting elements or searching a list, the user may optionally provide a compare function of the form:

```
int user_comp_name(PsmPartition partition, PsmAddress eltData,
                   void *dataBuffer);
```

When provided, this function is automatically called by the smlist function being invoked; when the

function is called it is passed the content of a list element (*eltData*, nominally the PsmAddress of an item in shared memory) and an argument, *dataBuffer*, which is nominally the address in local memory of some other item in the same format. The user-supplied function normally compares some key values of the two data items and returns 0 if they are equal, an integer less than 0 if *eltData*'s key value is less than that of *dataBuffer*, and an integer greater than 0 if *eltData*'s key value is greater than that of *dataBuffer*. These return values will produce a list in ascending order. If the user desires the list to be in descending order, the function must reverse the signs of these return values.

When deleting an element or destroying a list, the user may optionally provide a delete function of the form:

```
void user_delete_name(PsmPartition partition, PsmAddress eltData,
                      void *argData)
```

When provided, this function is automatically called by the smlist function being invoked; when the function is called it is passed the content of a list element (*eltData*, nominally the PsmAddress of an item in shared memory) and an argument, *argData*, which if non-NULL is normally the address in local memory of a data item providing context for the list element deletion. The user-supplied function performs any application-specific cleanup associated with deleting the element, such as freeing the element's content data item and/or other memory associated with the element.

EXAMPLE

For an example of the use of smlist, see the file smlistsh.c in the utils directory of ICI.

SEE ALSO

lyst (3), *platform* (3), *psm* (3)

NAME

zco – library for manipulating zero–copy objects

SYNOPSIS

```
#include "zco.h"

typedef enum
{
    ZcoFileSource = 1,
    ZcoSdrSource = 2,
    ZcoZcoSource = 3
} ZcoMedium;

typedef void (*ZcoCallback);

[see description for available functions]
```

DESCRIPTION

"Zero-copy objects" (ZCOs) are abstract data access representations designed to minimize I/O in the encapsulation of application source data within one or more layers of communication protocol structure. ZCOs are constructed within the heap space of an SDR to which implementations of all layers of the stack must have access. Each ZCO contains information enabling access to the source data objects, together with (a) a linked list of zero or more "extents" that reference portions of these source data objects and (b) linked lists of protocol header and trailer capsules that have been explicitly attached to the ZCO since its creation. The concatenation of the headers (in ascending stack sequence), source data object extents, and trailers (in descending stack sequence) is what is to be transmitted or has been received.

Each source data object may be either a file (identified by pathname stored in a "file reference" object in SDR heap) or an array of bytes in SDR heap space (identified by SDR address). Each protocol header or trailer capsule indicates the length and the address (within SDR heap space) of a single protocol header or trailer at some layer of the stack. Note that for some purposes the source data object for a newly added extent of a ZCO may be specified indirectly, by reference to an extent of an existing ZCO.

The extents of multiple ZCOs may reference the same files and/or SDR source data objects. The source data objects are reference-counted to ensure that they are deleted automatically when (and only when) all ZCO extents that reference them have been deleted.

Note that the safety of shared access to a ZCO is protected by the fact that the ZCO resides in SDR and therefore cannot be modified other than in the course of an SDR transaction, which serializes access. Moreover, extraction of data from a ZCO may entail the reading of file-based source data extents, which may cause file progress to be updated in one or more file reference objects in the SDR heap. For this reason, all ZCO "transmit" and "receive" functions must be performed within SDR transactions.

Note also that ZCO can more broadly be used as a general-purpose reference counting system for non-volatile data objects, where a need for such a system is identified.

The total volume of file system space that may be occupied by file-sourced ZCO extents and the total volume of SDR heap space that may be occupied by heap-sourced ZCO extents are system configuration parameters that may be set by ZCO library functions. Those limits are enforced when extents are appended to ZCOs: total ZCO file space occupancy and total ZCO heap occupancy are updated continuously as ZCOs are created and destroyed, and the formation of a new extent is prohibited when the length of the extent exceeds the difference between the applicable limit and the corresponding current occupancy total.

void zco_register_callback(ZcoCallback notify)

This function registers the "callback" function that the ZCO system will invoke every time a ZCO is destroyed, making ZCO file and/or heap space available for the formation of new ZCO extents. This mechanism can be used, for example, to notify tasks that are waiting for ZCO space to be made available so that they can resume some communication protocol procedure.

void zco_unregister_callback( )
This function simply unregisters the currently registered callback function for ZCO destruction.

Object zco_create_file_ref(Sdr sdr, char *pathName, char *cleanupScript)
Creates and returns a new file reference object, which can be used as the source data extent location for creating a ZCO whose source data object is the file identified by *pathName*. *cleanupScript*, if not NULL, is invoked at the moment the last ZCO that cites this file reference is destroyed [normally upon delivery either down to the ''ZCO transition layer'' of the protocol stack or up to a ZCO-capable application]. A zero-length string is interpreted as implicit direction to delete the referenced file when the file reference object is destroyed. Maximum length of *cleanupScript* is 255. Returns SDR location of file reference object on success, 0 on any error.

Object zco_revise_file_ref(Sdr sdr, Object fileRef, char *pathName, char *cleanupScript)
Changes the *pathName* and *cleanupScript* of the indicated file reference. The new values of these fields are validated as for *zco_create_file_ref()*. Returns 0 on success, –1 on any error.

char *zco_file_ref_path(Sdr sdr, Object fileRef, char *buffer, int buflen)
Retrieves the pathName associated with *fileRef* and stores it in *buffer*, truncating it to fit (as indicated by *buflen*) and NULL-terminating it. On success, returns *buffer*; returns NULL on any error.

int zco_file_ref_xmit_eof(Sdr sdr, Object fileRef)
Returns 1 if the last octet of the referenced file (as determined at the time the file reference object was created) has been read by ZCO via a reader with file offset tracking turned on. Otherwise returns zero.

void zco_destroy_file_ref(Sdr sdr, Object fileRef)
If the file reference object residing at location *fileRef* within the indicated Sdr is no longer in use (no longer referenced by any ZCO), destroys this file reference object immediately. Otherwise, flags this file reference object for destruction as soon as the last reference to it is removed.

vast zco_get_file_occupancy(Sdr sdr)
Returns the total number of file system space bytes occupied by ZCOs created in this Sdr.

void zco_set_max_file_occupancy(Sdr sdr, vast occupancy)
Declares the total number of file system space bytes that may be occupied by ZCOs created in this Sdr.

vast zco_get_max_file_occupancy(Sdr sdr)
Returns the total number of file system space bytes that may be occupied by ZCOs created in this Sdr.

int zco_enough_file_space(Sdr sdr, vast length)
Returns 1 if the total remaining file system space available for ZCOs in this Sdr is greater than *length*. Returns 0 otherwise.

vast zco_get_heap_occupancy(Sdr sdr)
Returns the total number of SDR heap space bytes occupied by ZCOs created in this Sdr.

void zco_set_max_heap_occupancy(Sdr sdr, vast occupancy)
Declares the total number of SDR heap space bytes that may be occupied by ZCOs created in this Sdr.

vast zco_get_max_heap_occupancy(Sdr sdr)
Returns the total number of SDR heap space bytes that may be occupied by ZCOs created in this Sdr.

int zco_enough_heap_space(Sdr sdr, vast length)
Returns 1 if the total remaining SDR heap space available for ZCOs in this Sdr is greater than *length*. Returns 0 otherwise.

Object zco_create(Sdr sdr, ZcoMedium firstExtentSourceMedium, Object firstExtentLocation, vast firstExtentOffset, vast firstExtentLength)
Creates a new ZCO. *firstExtentLocation* and *firstExtentLength* must either both be zero (indicating that *zco_append_extent()* will be used to insert the first source data extent later) or else both be non-zero. If *firstExtentLocation* is non-zero, then (a) *firstExtentLocation* must be the SDR location of a file reference object if *firstExtentSourceMedium* is ZcoFileSource and must otherwise be the SDR location of the source data itself, and (b) *firstExtentOffset* indicates how many leading bytes of the source data object should be skipped over when adding the initial source data extent to the new ZCO. On success,

returns the SDR location of the new ZCO. Returns 0 if there is insufficient ZCO space for creation of the new ZCO; returns ((Object) −1) on any error.

int zco_append_extent(Sdr sdr, Object zco, ZcoMedium sourceMedium, Object location, vast offset, vast length)

Appends the indicated source data extent to the indicated ZCO, as described for *zco_create()*. Both the *location* and *length* of the source data must be non-zero. Returns *length* on success, 0 if there is insufficient ZCO space for creation of the new source data extent, −1 on any error.

int zco_prepend_header(Sdr sdr, Object zco, char *header, vast length)
int zco_append_trailer(Sdr sdr, Object zco, char *trailer, vast length)
void zco_discard_first_header(Sdr sdr, Object zco)
void zco_discard_last_trailer(Sdr sdr, Object zco)

These functions attach and remove the ZCO's headers and trailers. *header* and *trailer* are assumed to be arrays of octets, not necessarily text. Attaching a header or trailer causes it to be written to the SDR. The prepend and append functions return 0 on success, −1 on any error.

void zco_destroy(Sdr sdr, Object zco)

Destroys the indicated Zco. This reduces the reference counts for all files and SDR objects referenced in the ZCO's extents, resulting in the freeing of SDR objects and (optionally) the deletion of files as those reference count drop to zero.

Object zco_clone(Sdr sdr, Object zco, vast offset, vast length)

Creates a new ZCO whose source data is a copy of a subset of the source data of the referenced ZCO. Portions of the source data extents of the original ZCO are copied as necessary, but no header or trailer capsules are copied. Returns SDR location of the new ZCO on success, 0 on any error.

vast zco_clone_source_data(Sdr sdr, Object toZco, Object fromZco, vast offset, vast length)

Appends to *toZco* a copy of a subset of the source data of *fromZCO*. Portions of the source data extents of *fromZCO* are copied as necessary. Returns *length* on success, −1 on any error.

vast zco_length(Sdr sdr, Object zco)

Returns length of entire ZCO, including all headers and trailers and all source data extents. This is the size of the object that would be formed by concatenating the text of all headers, trailers, and source data extents into a single serialized object.

vast zco_source_data_length(Sdr sdr, Object zco)

Returns length of entire ZCO minus the lengths of all attached header and trailer capsules. This is the size of the object that would be formed by concatenating the text of all source data extents (including those that are presumed to contain header or trailer text attached elsewhere) into a single serialized object.

void zco_start_transmitting(Object zco, ZcoReader *reader)

Used by underlying protocol layer to start extraction of an outbound ZCO's bytes (both from header and trailer capsules and from source data extents) for ''transmission'' — i.e., the copying of bytes into a memory buffer for delivery to some non-ZCO-aware protocol implementation. Initializes reading at the first byte of the total concatenated ZCO object. Populates *reader*, which is used to keep track of ''transmission'' progress via this ZCO reference.

Note that this function can be called multiple times to restart reading at the start of the ZCO. Note also that multiple ZcoReader objects may be used concurrently, by the same task or different tasks, to advance through the ZCO independently.

void zco_track_file_offset(ZcoReader *reader)

Turns on file offset tracking for this reader.

vast zco_transmit(Sdr sdr, ZcoReader *reader, vast length, char *buffer)

Copies *length* as-yet-uncopied bytes of the total concatenated ZCO (referenced by *reader*) into *buffer*. If *buffer* is NULL, skips over *length* bytes without copying. Returns the number of bytes copied (or skipped) on success, 0 on any file access error, −1 on any other error.

void zco_start_receiving(Object zco, ZcoReader *reader)
: Used by overlying protocol layer to start extraction of an inbound ZCO's bytes for "reception" — i.e., the copying of bytes into a memory buffer for delivery to a protocol header parser, to a protocol trailer parser, or to the ultimate recipient (application). Initializes reading of headers, source data, and trailers at the first byte of the concatenated ZCO objects. Populates *reader*, which is used to keep track of "reception" progress via this ZCO reference and is required.

vast zco_receive_headers(Sdr sdr, ZcoReader *reader, vast length, char *buffer)
: Copies *length* as-yet-uncopied bytes of presumptive protocol header text from ZCO source data extents into *buffer*. If *buffer* is NULL, skips over *length* bytes without copying. Returns number of bytes copied (or skipped) on success, 0 on any file access error, –1 on any other error.

void zco_delimit_source(Sdr sdr, Object zco, vast offset, vast length)
: Sets the computed offset and length of actual source data in the ZCO, thereby implicitly establishing the total length of the ZCO's concatenated protocol headers as *offset* and the location of the ZCO's innermost protocol trailer as the sum of *offset* and *length*. Offset and length are typically determined from the information carried in received presumptive protocol header text.

vast zco_receive_source(Sdr sdr, ZcoReader *reader, vast length, char *buffer)
: Copies *length* as-yet-uncopied bytes of source data from ZCO extents into *buffer*. If *buffer* is NULL, skips over *length* bytes without copying. Returns number of bytes copied (or skipped) on success, 0 on any file access error, –1 on any other error.

vast zco_receive_trailers(Sdr sdr, ZcoReader *reader, vast length, char *buffer)
: Copies *length* as-yet-uncopied bytes of trailer data from ZCO extents into *buffer*. If *buffer* is NULL, skips over *length* bytes without copying. Returns number of bytes copied (or skipped) on success, 0 on any file access error, –1 on any other error.

void zco_strip(Sdr sdr, Object zco)
: Deletes all source data extents that contain only header or trailer data and adjusts the offsets and/or lengths of all remaining extents to exclude any known header or trailer data. This function is useful when handling a ZCO that was received from an underlying protocol layer rather than from an overlying application or protocol layer; use it before starting the transmission of the ZCO to another node or before enqueuing it for reception by an overlying application or protocol layer.

SEE ALSO

sdr (3)

NAME

ltp – Licklider Transmission Protocol (LTP) communications library

SYNOPSIS

```
#include "ltp.h"

typedef enum
{
    LtpNoNotice = 0,
    LtpExportSessionStart,
    LtpXmitComplete,
    LtpExportSessionCanceled,
    LtpExportSessionComplete,
    LtpRecvGreenSegment,
    LtpRecvRedPart,
    LtpImportSessionCanceled
} LtpNoticeType;

[see description for available functions]
```

DESCRIPTION

The ltp library provides functions enabling application software to use LTP to send and receive information reliably over a long-latency link. It conforms to the LTP specification as documented by the Delay-Tolerant Networking Research Group of the Internet Research Task Force.

The LTP notion of **engine ID** corresponds closely to the Internet notion of a host, and in ION engine IDs are normally indistinguishable from node numbers (and from the ''element numbers'' in Bundle Protocol endpoint IDs conforming to the ''ipn'' scheme).

The LTP notion of **client ID** corresponds closely to the Internet notion of ''protocol number'' as used in the Internet Protocol. It enables data from multiple applications — clients — to be multiplexed over a single reliable link. However, for ION operations we normally use LTP exclusively for the transmission of Bundle Protocol data, identified by client ID = 1.

int *ltp_attach()*

Attaches the application to LTP functionality on the lcoal computer. Returns 0 on success, –1 on any error.

void *ltp_detach()*

Terminates all access to LTP functionality on the local computer.

int ltp_send(uvast destinationEngineId, unsigned int clientId, Object clientServiceData, unsigned int redLength, LtpSessionId *sessionId)

Sends a client service data unit to the application that is waiting for data tagged with the indicated *clientId* as received at the remote LTP engine identified by *destinationEngineId*.

clientServiceData must be a ''zero-copy object'' reference as returned by *zco_create()*. Note that LTP will privately make and destroy its own reference to the client service data object; the application is free to destroy its reference at any time.

redLength indicates the number of leading bytes of data in *clientServiceData* that are to be sent reliably, i.e., with selective retransmission in response to explicit or implicit negative acknowledgment as necessary. All remaining bytes of data in *clientServiceData* will be sent as ''green'' data, i.e., unreliably. If *redLength* is zero, the entire client service data unit will be sent unreliably. If the entire client service data unit is to be sent reliably, *redLength* may be simply be set to LTP_ALL_RED (i.e., –1).

On success, the function populates **sessionId* with the source engine ID and the ''session number'' assigned to transmission of this client service data unit and returns zero. The session number may be used to link future LTP processing events, such as transmission cancellation, to the affected client

service data. *ltp_send()* returns –1 on any error.

int ltp_open(unsigned int clientId)

Establishes the application's exclusive access to received service data units tagged with the indicated client service data ID. At any time, only a single application task is permitted to receive service data units for any single client service data ID.

Returns 0 on success, –1 on any error (e.g., the indicated client service is already being held open by some other application task).

int ltp_get_notice(unsigned int clientId, LtpNoticeType *type, LtpSessionId *sessionId, unsigned char *reasonCode, unsigned char *endOfBlock, unsigned int *dataOffset, unsigned int *dataLength, Object *data)

Receives notices of LTP processing events pertaining to the flow of service data units tagged with the indicated client service ID. The nature of each event is indicated by **type*. Additional parameters characterizing the event are returned in **sessionId*, **reasonCode*, **endOfBlock*, **dataOffset*, **dataLength*, and **data* as relevant.

The value returned in **data* is always a zero-copy object; use the zco_* functions defined in ''zco.h'' to retrieve the content of that object.

When the notice is an LtpRecvGreenSegment, the ZCO returned in **data* contains the content of a single LTP green segment. Reassembly of the green part of some block from these segments is the responsibility of the application.

When the notice is an LtpRecvRedPart, the ZCO returned in **data* contains the red part of a possibly aggregated block. The ZCO's content may therefore comprise multiple service data objects. Extraction of individual service data objects from the aggregated block is the responsibility of the application. A simple way to do this is to prepend the length of the service data object to the object itself (using zco_prepend_header) before calling ltp_send, so that the receiving application can alternate extraction of object lengths and objects from the delivered block's red part.

The cancellation of an export session may result in delivery of multiple LtpExportSessionCanceled notices, one for each service data unit in the export session's (potentially) aggregated block. The ZCO returned in **data* for each such notice is a service data unit ZCO that had previously been passed to *ltp_send()*.

ltp_get_notice() always blocks indefinitely until an LTP processing event is delivered.

Returns zero on success, –1 on any error.

void ltp_interrupt(unsigned int clientId)

Interrupts an *ltp_get_notice()* invocation. This function is designed to be called from a signal handler; for this purpose, *clientId* may need to be obtained from a static variable.

void ltp_release_data(Object data)

Releases the resources allocated to hold *data*, a client service data ZCO.

void ltp_close(unsigned int clientId)

Terminates the application's exclusive access to received service data units tagged with the indicated client service data ID.

SEE ALSO

ltpadmin (1), *ltprc* (5), *zco* (3)

NAME

acsrc – Aggregate Custody Signal management commands file

DESCRIPTION

Aggregate Custody Signal management commands are passed to **acsadmin** either in a file of text lines or interactively at **acsadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the Aggregate Custody Signal management commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by acsadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

1 <logLevel> [<heapWords>]
The **initialize** command. Until this command is executed, Aggregate Custody Signals are not in operation on the local ION node and most *acsadmin* commands will fail.

The *logLevel* argument specifies at which log level the ACS appending and transmitting implementation should record its activity to the ION log file. This argument is the bitwise ''OR'' of the following log levels:

0x01 ERROR
Errors in ACS programming are logged.

0x02 WARN
Warnings like ''out of memory'' that don't cause ACS to fail but may change behavior are logged.

0x04 INFO
Informative information like ''this custody signal is a duplicate'' is logged.

0x08 DEBUG
Verbose information like the state of the pending ACS tree is logged.

The optional *heapWords* argument informs ACS to allocate that many heap words in its own DRAM SDR for constructing pending ACS. If not supplied, the default ACS_SDR_DEFAULT_HEAPWORDS is used. Once all ACS SDR is allocated, any incoming custodial bundles that would trigger an ACS will trigger a normal, non-aggregate custody signal instead, until ACS SDR is freed. If your node intermittently emits non-aggregate custody signals when it should emit ACS, you should increase *heapWords*.

Since ACS uses SDR only for emitting Aggregate Custody Signals, ION can still receive ACS even if this command is not executed, or all ACS SDR memory is allocated.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

s <minimumCustodyId>
This command sets the minimum custody ID that the local bundle agent may use in custody transfer enhancement blocks that it emits. These custody IDs must be unique in the network (for the lifetime of the bundles to which they refer).

The *minimumCustodyId* provided is stored in SDR, and incremented every time a new custody ID is required. So, this command should be used only when the local bundle agent has discarded its SDR and restarted.

CUSTODIAN COMMANDS

a *custodianEid acsSize* [*acsDelay*]

The **add custodian** command. This command provides information about the ACS characteristics of a remote custodian. *custodianEid* is the custodian EID for which this command is providing information. *acsSize* is the preferred size of ACS bundles sent to *custodianEid*; ACS bundles this implementation sends to *custodianEid* will aggregate until ACS are at most *acsSize* bytes (if *acsSize* is smaller than 19 bytes, some ACS containing only one signal will exceed *acsSize* and be sent anyways; setting *acsSize* to 0 causes ''aggregates'' of only 1 signal to be sent).

acsDelay is the maximum amount of time to delay an ACS destined for this custodian before sending it, in seconds; if not specified, DEFAULT_ACS_DELAY will be used.

EXAMPLES

a ipn:15.0 100 27

Informs ACS on the local node that the local node should send ACS bundles destined for the custodian ipn:15.0 whenever they are 100 bytes in size or have been delayed for 27 seconds, whichever comes first.

SEE ALSO

acsadmin (1)

NAME

bprc – Bundle Protocol management commands file

DESCRIPTION

Bundle Protocol management commands are passed to **bpadmin** either in a file of text lines or interactively at **bpadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the Bundle Protocol management commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }

Echo control. Setting echo to 1 causes all output printed by bpadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed and the crypto suite BP was compiled with. HINT: combine with **e 1** command to log the version number at startup.

1 The **initialize** command. Until this command is executed, Bundle Protocol is not in operation on the local ION node and most *bpadmin* commands will fail.

r '*command_text*'

The **run** command. This command will execute *command_text* as if it had been typed at a console prompt. It is used to, for example, run another administrative program.

s The **start** command. This command starts all schemes and all protocols on the local node.

m heapmax *max_database_heap_per_acquisition*

The **manage heap for bundle acquisition** command. This command declares the maximum number of bytes of SDR heap space that will be occupied by any single bundle acquisition activity (nominally the acquisition of a single bundle, but this is at the discretion of the convergence-layer input task). All data acquired in excess of this limit will be written to a temporary file pending extraction and dispatching of the acquired bundle or bundles. Default is the size of a ZCO file reference object, the minimum SDR heap space occupancy in the event that all acquisition is into a file.

x The **stop** command. This command stops all schemes and all protocols on the local node.

w { 0 | 1 | *activity_spec* }

The **BP watch** command. This command enables and disables production of a continuous stream of user-selected Bundle Protocol activity indication characters. A watch parameter of ''1'' selects all BP activity indication characters; ''0'' de-selects all BP activity indication characters; any other *activity_spec* such as ''acz~'' selects all activity indication characters in the string, de-selecting all others. BP will print each selected activity indication character to **stdout** every time a processing event of the associated type occurs:

a new bundle is queued for forwarding

b bundle is queued for transmission

c bundle is popped from its transmission queue

m custody acceptance signal is received

w custody of bundle is accepted

x custody of bundle is refused

y bundle is accepted upon arrival

z bundle is queued for delivery to an application

~ bundle is abandoned (discarded) on attempt to forward it

! bundle is destroyed due to TTL expiration

& custody refusal signal is received

bundle is queued for re-forwarding due to CL protocol failure

j bundle is placed in ‘‘limbo’’ for possible future re-forwarding

k bundle is removed from ‘‘limbo’’ and queued for re-forwarding

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

SCHEME COMMANDS

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 a custodian endpoint identified within this scheme so that it can receive and process custody signals and bundle status reports.

c scheme *scheme_name* ’*forwarder_command*’ ’*admin_app_command*’

The **change scheme** command. This command sets the indicated scheme’s *forwarder_command* and *admin_app_command* to the strings provided as arguments.

d scheme *scheme_name*

The **delete scheme** command. This command deletes the scheme identified by *scheme_name*. The command will fail if any bundles identified in this scheme are pending forwarding, transmission, or delivery.

i scheme *scheme_name*

This command will print information (number and commands) about the endpoint naming scheme identified by *scheme_name*.

l scheme

This command lists all declared endpoint naming schemes.

s scheme *scheme_name*

The **start scheme** command. This command starts the forwarder and administrative endpoint tasks for the indicated scheme task on the local node.

x scheme *scheme_name*

The **stop scheme** command. This command stops the forwarder and administrative endpoint tasks for the indicated scheme task on the local node.

ENDPOINT COMMANDS

a endpoint *endpoint_ID* { q | x } [’*recv_script*’]

The **add endpoint** command. This command establishes a DTN endpoint named *endpoint_ID* on the local node. The remaining parameters indicate what is to be done when bundles destined for this endpoint arrive at a time when no application has got the endpoint open for bundle reception. If ’x’, then such bundles are to be discarded silently and immediately. If ’q’, then such bundles are to be enqueued for later delivery and, if *recv_script* is provided, *recv_script* is to be executed.

c endpoint *endpoint_ID* { q | x } [’*recv_script*’]

The **change endpoint** command. This command changes the action that is to be taken when bundles destined for this endpoint arrive at a time when no application has got the endpoint open for bundle reception, as described above.

d endpoint *endpoint_ID*

The **delete endpoint** command. This command deletes the endpoint identified by *endpoint_ID*. The command will fail if any bundles are currently pending delivery to this endpoint.

i endpoint *endpoint_ID*
This command will print information (disposition and script) about the endpoint identified by *endpoint_ID*.

l endpoint
This command lists all local endpoints, regardless of scheme name.

PROTOCOL COMMANDS

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.

d protocol *protocol_name*
The **delete protocol** command. This command deletes the convergence layer protocol identified by *protocol_name*. The command will fail if any ducts are still locally declared for this protocol.

i protocol *protocol_name*
This command will print information about the convergence layer protocol identified by *protocol_name*.

l protocol
This command lists all convergence layer protocols that can currently be utilized at the local node.

s protocol *protocol_name*
The **start protocol** command. This command starts all induct and outduct tasks for inducts and outducts that have been defined for the indicated CL protocol on the local node.

x protocol *protocol_name*
The **stop protocol** command. This command stops all induct and outduct tasks for inducts and outducts that have been defined for the indicated CL protocol on the local node.

INDUCT COMMANDS

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.

c induct *protocol_name duct_name* '*CLI_command*'
The **change induct** command. This command changes the command that is used to initiate operation of the induct task for the indicated duct.

d induct *protocol_name duct_name*
The **delete induct** command. This command deletes the induct identified by *protocol_name* and *duct_name*. The command will fail if any bundles are currently pending acquisition via this induct.

i induct *protocol_name duct_name*
This command will print information (the CLI command) about the induct identified by *protocol_name* and *duct_name*.

l induct [*protocol_name*]
If *protocol_name* is specified, this command lists all inducts established locally for the indicated CL protocol. Otherwise it lists all locally established inducts, regardless of protocol.

s induct *protocol_name duct_name*
The **start induct** command. This command starts the indicated induct task as defined for the indicated CL protocol on the local node.

x induct *protocol_name duct_name*
The **stop induct** command. This command stops the indicated induct task as defined for the indicated CL protocol on the local node.

OUTDUCT COMMANDS

a outduct *protocol_name duct_name* '*CLO_command*' [*max_payload_length*]
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 value of zero for *max_payload_length* indicates that bundles of any size can be accommodated; this is the default.

c outduct *protocol_name duct_name* '*CLO_command*' [*max_payload_length*]
The **change outduct** command. This command sets new values for the indicated duct's payload size limit and the command that is used to initiate operation of the outduct task for this duct.

d outduct *protocol_name duct_name*
The **delete outduct** command. This command deletes the outduct identified by *protocol_name* and *duct_name*. The command will fail if any bundles are currently pending transmission via this outduct.

i outduct *protocol_name duct_name*
This command will print information (the CLO command) about the outduct identified by *protocol_name* and *duct_name*.

l outduct [*protocol_name*]
If *protocol_name* is specified, this command lists all outducts established locally for the indicated CL protocol. Otherwise it lists all locally established outducts, regardless of protocol.

s outduct *protocol_name duct_name*
The **start outduct** command. This command starts the indicated outduct task as defined for the indicated CL protocol on the local node.

b outduct *protocol_name duct_name*
The **block outduct** command. This command disables transmission of bundles via the indicated outduct and reforwards all non-critical bundles currently queued for transmission via this outduct.

u outduct *protocol_name duct_name*
The **unblock outduct** command. This command re-enables transmission of bundles via the indicated outduct and reforwards all bundles in ''limbo'' in the hope that the unblocking of this outduct will enable some of them to be transmitted.

x outduct *protocol_name duct_name*
The **stop outduct** command. This command stops the indicated outduct task as defined for the indicated CL protocol on the local node.

EXAMPLES

a scheme ipn 'ipnfw' 'ipnadminep'
Declares the ''ipn'' scheme on the local node.

a protocol udp 1400 100 16384
Establishes access to the ''udp'' convergence layer protocol on the local node, estimating the number of payload bytes per ultimate (lowest-layer) frame to be 1400 with 100 bytes of total overhead (BP, UDP, IP, AOS) per lowest-layer frame, and setting the default nominal data rate to be 16384 bytes per second.

r 'ipnadmin flyby.ipnrc'
Runs the administrative program *ipnadmin* from within *bpadmin*.

SEE ALSO

bpadmin (1), *ipnadmin* (1), *dtn2admin* (1)

NAME

bssrc – IPN scheme configuration commands file adapted for Bundle Streaming Service

DESCRIPTION

IPN scheme configuration commands are passed to **bssadmin** either in a file of text lines or interactively at **bssadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line.

IPN scheme configuration commands (a) manage a table of destination endpoints that are known to be associated with Bundle Streaming Service (BSS) applications, (b) establish BSS-adapted egress plans for direct transmission to neighboring nodes that are members of endpoints identified in the ''ipn'' URI scheme, and (c) establish static default routing rules for forwarding bundles to specified destination nodes.

A BSS endpoint table **entry** identifies an IPN endpoint ID — in which the node number and/or service number may be the wild-card character '*' — that is known to be associated with a BSS application. These table entries enable **bssfw** to distinguish BSS bundles from non-BSS traffic and apply BSS-specific egress planning logic to the former while handling the latter in exactly the same way as **ipnfw**.

The egress **plan** established for a given neighboring node associates three default egress **duct expressions** with that node: one for BSS traffic that must be forwarded as real-time streaming data (using a convergence-layer protocol that does not perform retransmission), one for BSS traffic that must be forwarded as playback data (using a reliable convergence-layer protocol), and one for non-BSS traffic. These default duct expressions may be overridden by more narrowly scoped **planrule**s in specific circumstances: different egress duct expressions may apply when the source endpoint for the subject bundle identifies a specific node, a specific service, or both.

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 circumstances that characterize a specific rule within a general plan are expressed in a **qualifier**, a string of the form "*source_service_number source_node_number*" where either *source_service_number* or *source_node_number* may be an asterisk character (*) signifying ''all''.

Note that egress plans **must** be established for all neighboring nodes, regardless of whether or not contact graph routing is used for computing dynamic routes to distant nodes. This is by definition: if there isn't an egress plan to a node, it can't be considered a neighbor.

Static default routes are expressed as **groups** in the ipn-scheme routing database. 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 gateway node for any group is identified by an endpoint ID, which might or might not be an ipn-scheme EID; regardless, directing a bundle to the gateway for a group causes the bundle to be re-forwarded to that intermediate destination endpoint. Multiple groups may encompass the same node number, in which case the gateway associated with the most restrictive group (the one with the smallest range) is always selected.

The formats and effects of the BSS forwarding configuration commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }

Echo control. Setting echo to 1 causes all output printed by bssadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

ENTRY COMMANDS

a entry *service_nbr node_nbr*

The **add entry** command. This asserts that all bundles whose destination endpoint ID matches *service_nbr* and *node_nbr* (either or both of which may be the wild-card character '*') are to be processed as BSS traffic.

d entry *service_nbr node_nbr*

The **delete entry** command. This command rescinds a prior BSS assertion characterized by the exact same *service_nbr* and *node_nbr*.

l entry

This command lists all entries in the node's table of destination endpoint IDs that indicate BSS traffic.

PLAN COMMANDS

a plan *node_nbr non−BSS_duct_expression BSS_non−reliable_duct_expression BSS_reliable_duct_expression custody_expiration_interval*

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*. *custody_expiration_interval* indicates the number of seconds the BP agent must wait for custody acceptance after transmitting a bundle on either BSS duct before automatically re-forwarding the bundle. A general plan must be in place for a node before any more specific rules are declared.

c plan *node_nbr non−BSS_duct_expression BSS_non−reliable_duct_expression BSS_reliable_duct_expression custody_expiration_interval*

The **change plan** command. This command changes the duct expressions and/or custody expiration interval for the indicated plan.

d plan *node_nbr*

The **delete plan** command. This command deletes the egress plan for the node identified by *node_nbr*, including all associated rules.

i plan *node_nbr*

This command will print information (the default duct expressions, custody expiration interval, and all specific rules) about the egress plan for the node identified by *node_nbr*.

l plan

This command lists all egress plans established in the BSS database for the local node.

PLANRULE COMMANDS

a planrule *node_nbr qualifier non−BSS_duct_expression BSS_non−reliable_duct_expression BSS_reliable_duct_expression*

The **add planrule** command. This command establishes a planrule, i.e., a set of duct expressions that override the default duct expressions of the egress plan for the node identified by *node_nbr* in the event that the source endpoint ID of the subject bundle matches *qualifier*.

c planrule *node_nbr qualifier non−BSS_duct_expression BSS_non−reliable_duct_expression BSS_reliable_duct_expression*

The **change planrule** command. This command changes the duct expressions for the indicated planrule.

d planrule *node_nbr qualifier*

The **delete planrule** command. This command deletes the planrule identified by *node_nbr* and *qualifier*.

i planrule *node_nbr qualifier*
This command will print information (the duct expressions) about the planrule identified by *node_nbr* and *qualifier*.

l planrule *node_nbr*
This command lists all planrules in the plan for the indicated node.

GROUP COMMANDS

a group *first_node_nbr last_node_nbr gateway_endpoint_ID*
The **add group** command. This command establishes a ''group'' for static default routing as described above.

c group *first_node_nbr last_node_nbr gateway_endpoint_ID*
The **change group** command. This command changes the gateway node number for the group identified by *first_node_nbr* and *last_node_nbr* .

d group *first_node_nbr last_node_nbr*
The **delete group** command. This command deletes the group identified by *first_node_nbr* and *last_node_nbr*.

i group *first_node_nbr last_node_nbr*
This command will print information (the gateway endpoint ID) about the group identified by *first_node_nbr* and *last_node_nbr*.

l group
This command lists all groups defined in the BSS database for the local node.

GROUPRULE COMMANDS

a grouprule *first_node_nbr last_node_nbr qualifier gateway_endpoint_ID*
The **add grouprule** command. This command establishes a grouprule, i.e., a gateway endpoint ID that overrides the default gateway endpoint ID of the group identified by *first_node_nbr* and *last_node_nbr* in the event that the source endpoint ID of the subject bundle matches *qualifier*.

c grouprule *first_node_nbr last_node_nbr qualifier gateway_endpoint_ID*
The **change grouprule** command. This command changes the gateway EID for the indicated grouprule.

d grouprule *first_node_nbr last_node_nbr qualifier*
The **delete grouprule** command. This command deletes the grouprule identified by *first_node_nbr*, *last_node_nbr*, and *qualifier*.

i grouprule *first_node_nbr last_node_nbr qualifier*
This command will print information (the duct expression) about the grouprule identified by *node_nbr*, *last_node_nbr*, and *qualifier*.

l grouprule *first_node_nbr last_node_nbr*
This command lists all grouprules for the indicated group.

EXAMPLES

a plan 18 tcp/saturn.nasa.gov:5011 udp/*,saturn.nasa.gov:5012 tcp/saturn.nasa.gov:5011 3
Declares the egress plan to use for transmission from the local node to neighboring node 18. Any bundle for which the computed ''next hop'' node is node 18 will be queued for transmission to Internet host saturn.nasa.gov, using udp if the bundle is real-time BSS traffic and tcp otherwise; for BSS traffic, custodial retransmission will be initiated after 3 seconds if no custody acknowledgment is received.

a planrule 18 * 9 tcp/saturn.nasa.gov:5011 udp/*,saturn.nasa.gov:5012 tcp/neptune.nasa.gov:5011
Declares an egress plan override that applies to transmission to node 18 of any bundle whose source is node 9, regardless of the service that was the source of the bundle. Each such bundle must be queued for transmission to Internet host neptune.nasa.gov, rather than default host saturn.nasa.gov, if it is non-real-time BSS traffic.

a group 1 999 dtn://stargate

Declares a default route for bundles destined for all nodes whose numbers are in the range 1 through 999 inclusive: absent any other routing decision, such bundles are to be forwarded to ‘‘dtn://stargate’’.

SEE ALSO

bssadmin (1)

NAME

dtn2rc – "dtn" scheme configuration commands file

DESCRIPTION

''dtn'' scheme configuration commands are passed to **dtn2admin** either in a file of text lines or interactively at **dtn2admin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line.

''dtn'' scheme configuration commands mainly establish static routing rules for forwarding bundles to ''dtn''–scheme destination endpoints, identified by node names and demux names.

Static routes are expressed as **plan**s in the ''dtn''–scheme routing database. A plan that is established for a given node name associates a default routing **directive** with the named node, and that default directive may be overridden by more narrowly scoped **rule**s in specific circumstances: a different directive may apply when the destination endpoint ID specifies a particular demux name.

Each directive is a string of one of two possible forms:

f *endpoint_ID*

...or...

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

The former form signifies that the bundle is to be forwarded to the indicated endpoint, requiring that it be re-queued for processing by the forwarder for that endpoint (which might, but need not, be identified by another ''dtn''–scheme endpoint ID). The latter form signifies 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 node names and demux names cited in dtn2rc plans and overriding rules may be ''wild-carded''. That is, when the last character of a node name is either '*' or '~' (these two wild-card characters are equivalent for this purpose), the plan or rule applies to all nodes whose names are identical to the wild-carded node name up to the wild-card character; wild-carded demux names function in the same way. For example, a bundle whose destination EID's node name is ''//foghorn'' would be routed by plans citing the following node names: ''//foghorn'', ''//fogh*'', ''//fog~'', ''//*''. When multiple plans are all applicable to the same destination EID, the one citing the longest (i.e., most narrowly targeted) node name will be applied; when multiple rules overriding the same plan are all applicable to the same destination EID, the one citing the longest demux name will be applied.

The formats and effects of the DTN scheme configuration commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by dtn2admin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

PLAN COMMANDS

a plan *node_name default_directive*
The **add plan** command. This command establishes a static route for the bundles destined for the node identified by *node_name*. A general plan must be in place for a node before any more specific routing rules are declared.

d plan *node_name*
The **delete plan** command. This command deletes the static route for the node identified by *node_name*, including all associated rules.

i plan *node_name*
This command will print information (the default directive and all specific rules) about the static route for the node identified by *node_name*.

l plan
This command lists all static routes established in the DTN database for the local node.

RULE COMMANDS

a rule *node_name demux_name directive*
The **add rule** command. This command establishes a rule, i.e., a directive that overrides the default directive of the plan for the node identified by *node_name* in the event that the demux name of the subject bundle's destination endpoint ID matches *demux_name*.

c rule *node_name demux_name directive*
The **change rule** command. This command changes the directive for the indicated rule.

d rule *node_name demux_name*
The **delete rule** command. This command deletes the rule identified by *node_name* and *demux_name*.

i rule *node_name demux_name*
This command will print information (the directive) about the rule identified by *node_name* and *demux_name*.

l rule *node_name*
This command lists all rules in the plan for the indicated node.

EXAMPLES

a plan //bbn2 f ipn:8.41
Declares a static route from the local node to node "//bbn2". By default, any bundle destined for any endpoint whose node name is "//bbn2" will be forwarded to endpoint "ipn:8.41".

a plan //mitre1 x ltp/6
Declares a static route from the local node to node "//mitre1". By default, any bundle destined for any endpoint whose node name is "mitre1" will be queued for transmission on LTP outduct 6.

a rule //mitre1 fwd x ltp/18
Declares an overriding static routing rule for any bundle destined for node "//mitre1" whose destination demux name is "fwd". Each such bundle must be queued for transmission on LTP outduct 18 rather than the default (LTP outduct 6).

SEE ALSO

dtn2admin (1)

NAME

imcrc – IMC scheme configuration commands file

DESCRIPTION

IMC scheme configuration commands are passed to **ipnadmin** either in a file of text lines or interactively at **ipnadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line.

IMC scheme configuration commands simply establish which nodes are the local node's parents and children within a single IMC multicast tree. This single spanning tree, an overlay on a single BP-based network, is used to convey all multicast group membership assertions and cancellations in the network, for all groups. Each node privately tracks which of its immediate ''relatives'' in the tree are members of which multicast groups and on this basis selectively forwards — directly, to all (and only) interested relatives — the bundles destined for the members of each group.

Note that all of a node's immediate relatives in the multicast tree **must** be among its immediate neighbors in the underlying network. This is because multicast bundles can only be correctly forwarded within the tree if each forwarding node knows the identity of the relative that passed the bundle to it, so that the bundle is not passed back to that relative creating a routing loop. The identity of that prior forwarding node can only be known if the forwarding node was a neighbor, because no prior forwarding node (aside from the source) other than the immediate proximate (neighboring) sender of a received bundle is ever known.

IMC group IDs are unsigned integers, just as IPN node IDs are unsigned integers. The members of a group are nodes identified by node number, and the multicast tree parent and children of a node are neighboring nodes identified by node number.

The formats and effects of the IMC scheme configuration commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by ipnadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

KINSHIP COMMANDS

a *node_nbr* { 1 | 0 }
The **add kin** command. This command adds the neighboring node identified by *node_nbr* as an immediate relative of the local node. The Boolean value that follows the node number indicates whether or not this node is the local node's parent within the tree.

c *node_nbr* { 1 | 0 }
The **change kin** command. This command changes the parentage status of the indicated relative according to Boolean value that follows the node number, as noted for the **add kin** command.

d *node_nbr*
The **delete kin** command. This command deletes the immediate multicast tree relative identified by *node_nbr*. That node still exists but it is no longer a parent or child of the local node.

i *node_nbr*
This command will print information (the parentage switch) for the multicast tree relative identified by *node_nbr*.

l This command lists all of the local node's multicast tree relatives, indicating which one is its parent in the tree.

EXAMPLES

a 983 1

Declares that 983 is the local node's parent in the network's multicast tree.

SEE ALSO

imcadmin (1)

NAME

ipnrc – IPN scheme configuration commands file

DESCRIPTION

IPN scheme configuration commands are passed to **ipnadmin** either in a file of text lines or interactively at **ipnadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line.

IPN scheme configuration commands (a) establish egress plans for direct transmission to neighboring nodes that are members of endpoints identified in the ''ipn'' URI scheme and (b) establish static default routing rules for forwarding bundles to specified destination nodes.

The egress **plan** established for a given node associates a default egress **duct expression** with that node, and that default duct expression may be overridden by more narrowly scoped **planrule**s in specific circumstances: a different egress duct expression may apply when the source endpoint for the subject bundle identifies a specific node, a specific service, or both.

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 circumstances that characterize a specific rule within a general plan are expressed in a **qualifier**, a string of the form "*source_service_number source_node_number*" where either *source_service_number* or *source_node_number* may be an asterisk character (*) signifying ''all''.

Note that egress plans **must** be established for all neighboring nodes, regardless of whether or not contact graph routing is used for computing dynamic routes to distant nodes. This is by definition: if there isn't an egress plan to a node, it can't be considered a neighbor.

Static default routes are expressed as **groups** in the ipn-scheme routing database. 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 gateway node for any group is identified by an endpoint ID, which might or might not be an ipn-scheme EID; regardless, directing a bundle to the gateway for a group causes the bundle to be re-forwarded to that intermediate destination endpoint. Multiple groups may encompass the same node number, in which case the gateway associated with the most restrictive group (the one with the smallest range) is always selected.

The formats and effects of the IPN scheme configuration commands are described below.

GENERAL COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by ipnadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

PLAN COMMANDS

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.

c plan *node_nbr default_duct_expression*
The **change plan** command. This command changes the default duct expression for the indicated plan.

d plan *node_nbr*
The **delete plan** command. This command deletes the egress plan for the node identified by *node_nbr*, including all associated rules.

i plan *node_nbr*
This command will print information (the default duct expression and all specific rules) about the egress plan for the node identified by *node_nbr*.

l plan
This command lists all egress plans established in the IPN database for the local node.

PLANRULE COMMANDS

a planrule *node_nbr qualifier duct_expression*
The **add planrule** command. This command establishes a planrule, i.e., a duct expression that overrides the default duct expression of the egress plan for the node identified by *node_nbr* in the event that the source endpoint ID of the subject bundle matches *qualifier*.

c planrule *node_nbr qualifier duct_expression*
The **change planrule** command. This command changes the duct expression for the indicated planrule.

d planrule *node_nbr qualifier*
The **delete planrule** command. This command deletes the planrule identified by *node_nbr* and *qualifier*.

i planrule *node_nbr qualifier*
This command will print information (the duct expression) about the planrule identified by *node_nbr* and *qualifier*.

l planrule *node_nbr*
This command lists all planrules in the plan for the indicated node.

GROUP COMMANDS

a group *first_node_nbr last_node_nbr gateway_endpoint_ID*
The **add group** command. This command establishes a ''group'' for static default routing as described above.

c group *first_node_nbr last_node_nbr gateway_endpoint_ID*
The **change group** command. This command changes the gateway node number for the group identified by *first_node_nbr* and *last_node_nbr* .

d group *first_node_nbr last_node_nbr*
The **delete group** command. This command deletes the group identified by *first_node_nbr* and *last_node_nbr*.

i group *first_node_nbr last_node_nbr*
This command will print information (the gateway endpoint ID) about the group identified by *first_node_nbr* and *last_node_nbr*.

l group
This command lists all groups defined in the IPN database for the local node.

GROUPRULE COMMANDS

a grouprule *first_node_nbr last_node_nbr qualifier gateway_endpoint_ID*
The **add grouprule** command. This command establishes a grouprule, i.e., a gateway endpoint ID that overrides the default gateway endpoint ID of the group identified by *first_node_nbr* and *last_node_nbr* in the event that the source endpoint ID of the subject bundle matches *qualifier*.

c grouprule *first_node_nbr last_node_nbr qualifier gateway_endpoint_ID*
The **change grouprule** command. This command changes the gateway EID for the indicated grouprule.

d grouprule *first_node_nbr last_node_nbr qualifier*
The **delete grouprule** command. This command deletes the grouprule identified by *first_node_nbr*, *last_node_nbr*, and *qualifier*.

i grouprule *first_node_nbr last_node_nbr qualifier*
This command will print information (the duct expression) about the grouprule identified by *node_nbr*, *last_node_nbr*, and *qualifier*.

l grouprule *first_node_nbr last_node_nbr*
This command lists all grouprules for the indicated group.

EXAMPLES

a plan 18 ltp/18
Declares the egress plan to use for transmission from the local node to neighboring node 18. Any bundle for which the computed ‘‘next hop’’ node is node 18 will be queued for transmission on LTP outduct 18.

a planrule 18 * 9 ltp/−18
Declares an egress plan override that applies to transmission to node 18 of any bundle whose source is node 9, regardless of the service that was the source of the bundle. Each such bundle must be queued for unreliable transmission on LTP outduct 18 rather than the default (standard transmission on LTP outduct 18).

a group 1 999 dtn://stargate
Declares a default route for bundles destined for all nodes whose numbers are in the range 1 through 999 inclusive: absent any other routing decision, such bundles are to be forwarded to ‘‘dtn://stargate’’.

SEE ALSO

ipnadmin (1)

NAME

lgfile – ION Load/Go source file

DESCRIPTION

The ION Load/Go system enables the execution of ION administrative programs at remote nodes:

The **lgsend** program reads a Load/Go source file from a local file system, encapsulates the text of that source file in a bundle, and sends the bundle to a designated DTN endpoint on the remote node.

An **lgagent** task running on the remote node, which has opened that DTN endpoint for bundle reception, receives the extracted payload of the bundle — the text of the Load/Go source file — and processes it.

Load/Go source file content is limited to newline-terminated lines of ASCII characters. More specifically, the text of any Load/Go source file is a sequence of *line sets* of two types: *file capsules* and *directives*. Any Load/Go source file may contain any number of file capsules and any number of directives, freely intermingled in any order, but the typical structure of a Load/Go source file is simply a single file capsule followed by a single directive.

Each *file capsule* is structured as a single start-of-capsule line, followed by zero or more capsule text lines, followed by a single end-of-capsule line. Each start-of-capsule line is of this form:

[*file_name*

Each capsule text line can be any line of ASCII text that does not begin with an opening ([) or closing (]) bracket character.

A text line that begins with a closing bracket character (]) is interpreted as an end-of-capsule line.

A *directive* is any line of text that is not one of the lines of a file capsule and that is of this form:

!*directive_text*

When **lgagent** identifies a file capsule, it copies all of the capsule's text lines to a new file named *file_name* that it creates in the current working directory. When **lgagent** identifies a directive, it executes the directive by passing *directive_text* to the *pseudoshell()* function (see *platform* (3)). **lgagent** processes the line sets of a Load/Go source file in the order in which they appear in the file, so the *directive_text* of a directive may reference a file that was created as the result of processing a prior file capsule line set in the same source file.

Note that lgfile directives are passed to *pseudoshell()*, which on a VxWorks platform will always spawn a new task; the first argument in *directive_text* must be a symbol that VxWorks can resolve to a function, not a shell command. Also note that the arguments in *directive_text* will be actual task arguments, not shell command-line arguments, so they should never be enclosed in double-quote characters ("). However, any argument that contains embedded whitespace must be enclosed in single-quote characters (') so that *pseudoshell()* can parse it correctly.

EXAMPLES

Presenting the following lines of source file text to **lgsend**:

[cmd33.bprc

x protocol ltp

]

!bpadmin cmd33.bprc

should cause the receiving node to halt the operation of the LTP convergence-layer protocol.

SEE ALSO

lgsend (1), *lgagent* (1), *platform* (3)

NAME

cfdprc – CCSDS File Delivery Protocol management commands file

DESCRIPTION

CFDP management commands are passed to **cfdpadmin** either in a file of text lines or interactively at **cfdpadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the CFDP management commands are described below.

COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by cfdpadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

1 The **initialize** command. Until this command is executed, CFDP is not in operation on the local ION node and most *cfdpadmin* commands will fail.

i The **info** This command will print information about the current state of the local CFDP entity, including the current settings of all parameters that can be managed as described below.

s '*UTS command*'
The **start** command. This command starts the UT-layer service task for the local CFDP entity.

m discard { 0 | 1 }
The **manage discard** command. This command enables or disables the discarding of partially received files upon cancellation of a file reception.

m requirecrc { 0 | 1 }
The **manage CRC data integrity** command. This command enables or disables the attachment of CRCs to all PDUs issued by the local CFDP entity.

m fillchar *file_fill_character*
The **manage fill character** command. This command establishes the fill character to use for the portions of an incoming file that have not yet been received. The fill character is normally expressed in hex, e.g., 0xaa.

m ckperiod *check_cycle_period*
The **manage check interval** command. This command establishes the number of seconds following reception of the EOF PDU — or following expiration of a prior check cycle — after which the local CFDP will check for completion of a file that is being received.

m maxtimeouts *check_cycle_limit*
The **manage check limit** command. This command establishes the number of check cycle expirations after which the local CFDP entity will invoke the check cycle expiration fault handler upon expiration of a check cycle.

m maxtrnbr *max_transaction_number*
The **manage transaction numbers** command. This command establishes the largest possible transaction number used by the local CFDP entity for file transmission transactions. After this number has been used, the transaction number assigned to the next transaction will be 1.

m segsize *max_bytes_per_file_data_segment*
The **manage segment size** command. This command establishes the number of bytes of file data in each file data PDU transmitted by the local CFDP entity in the absence of an application-supplied reader function.

m inactivity *inactivity_period*

The **manage inactivity period** command. This command establishes the number of seconds that a CFDP file transfer is allowed to go idle before being canceled for inactivity. The default is one day.

x The **stop** command. This command stops the UT-layer service task for the local CFDP engine.

w { 0 | 1 | <activity_spec> }

The **CFDP watch** command. This command enables and disables production of a continuous stream of user-selected CFDP activity indication characters. A watch parameter of ''1'' selects all CFDP activity indication characters; ''0'' de-selects all CFDP activity indication characters; any other *activity_spec* such as ''p'' selects all activity indication characters in the string, de-selecting all others. CFDP will print each selected activity indication character to **stdout** every time a processing event of the associated type occurs:

p CFDP PDU transmitted

q CFDP PDU received

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

EXAMPLES

m requirecrc 1

Initiates attachment of CRCs to all subsequently issued CFDP PDUs.

SEE ALSO

cfdpadmin (1), *bputa* (1)

NAME

ionconfig – ION node configuration parameters file

DESCRIPTION

ION node configuration parameters are passed to **ionadmin** in a file of parameter name/value pairs:

parameter_name parameter_value

Any line of the file that begins with a '#' character is considered a comment and is ignored.

ionadmin supplies default values for any parameters for which no value is provided in the node configuration parameters file.

The applicable parameters are as follows:

sdrName
: This is the character string by which this ION node's SDR database will be identified. (Note that the SDR database infrastructure enables multiple databases to be constructed on a single host computer.) The default value is ''ion''.

configFlags
: This is the bitwise ''OR'' (i.e., the sum) of the flag values that characterize the SDR database to use for this ION node. The default value is 1. The SDR configuration flags are documented in detail in *sdr* (3). To recap:

  SDR_IN_DRAM (1)
  : The SDR is implemented in a region of shared memory. [Possibly with write-through to a file, for fault tolerance.]

  SDR_IN_FILE (2)
  : The SDR is implemented as a file. [Possibly cached in a region of shared memory, for faster data retrieval.]

  SDR_REVERSIBLE (4)
  : Transactions in the SDR are written ahead to a log, making them reversible.

  SDR_BOUNDED (8)
  : SDR heap updates are not allowed to cross object boundaries.

heapKey
: This is the shared-memory key by which the pre-allocated block of shared dynamic memory to be used as heap space for this SDR can be located, if applicable. The default value is –1, i.e., not specified and not applicable.

pathName
: This is the fully qualified path name of the directory in which are located (a) the file to be used as heap space for this SDR (which will be created, if it doesn't already exist), in the event that the SDR is to be implemented in a file, and (b) the file to be used to log the database updates of each SDR transaction, in the event that transactions in this SDR are to be reversible. The default value is **/usr/ion**.

heapWords
: This is the number of words (of 32 bits each on a 32–bit machine, 64 bits each on a 64–bit machine) of nominally non-volatile storage to use for ION's SDR database. If the SDR is to be implemented in shared memory and no *heapKey* is specified, a block of shared memory of this size will be allocated (e.g., by *malloc()*) at the time the node is created. If the SDR is to be implemented in a file and no file named **ion.sdr** exists in the directory identified by *pathName*, then a file of this name and size will be created in this directory and initialized to all binary zeroes. The default value is 250000 words (1 million bytes on a 32–bit computer).

wmKey
: This is the shared-memory key by which this ION node's working memory will be identified. The default value is 65537.

wmAddress

This is the address of the block of dynamic memory — volatile storage, which is not expected to persist across a system reboot — to use for this ION node's working memory. If zero, the working memory block will be allocated from system memory (e.g., by *malloc()*) at the time the local ION node is created. The default value is zero.

wmSize

This is the size of the block of dynamic memory that will be used for this ION node's working memory. If *wmAddress* is zero, a block of system memory of this size will be allocated (e.g., by *malloc()*) at the time the node is created. The default value is 5000000 (5 million bytes).

EXAMPLE

configFlags 1

heapWords 2500000

heapKey –1

pathName 'usr/ion'

wmSize 5000000

wmAddress 0

SEE ALSO

ionadmin (1)

NAME

ionrc – ION node management commands file

DESCRIPTION

ION node management commands are passed to **ionadmin** either in a file of text lines or interactively at **ionadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the ION node management commands are described below.

TIME REPRESENTATION

For many ION node management commands, time values must be passed as arguments. Every time value may be represented in either of two formats. Absolute time is expressed as:

yyyy/*mm*/*dd*–*hh*:*mm*:*ss*

Relative time (a number of seconds following the current *reference time*, which defaults to the current time at the moment *ionadmin* began execution but which can be overridden by the **at** command described below) is expressed as:

+*ss*

COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by ionadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

1 *node_number* { *ion_config_filename* | '' }
The **initialize** command. Until this command is executed, the local ION node does not exist and most *ionadmin* commands will fail.

The command configures the local node to be identified by *node_number*, a CBHE node number which uniquely identifies the node in the delay-tolerant network. It also configures ION's data store (SDR) and shared working-memory region using either the settings found in the *ion_config_filename* file or, if '' is supplied as the *ion_config_filename*, a set of default settings. Please see *ionconfig* (5) for details.

For example:

```
1 19 ''
```

would initialize ION on the local computer, assigning the local ION node the node number 19 and using default values to configure the data store and shared working-memory region.

@ *time*
The **at** command. This is used to set the reference time that will be used for interpreting relative time values from now until the next revision of reference time. Note that the new reference time can be a relative time, i.e., an offset beyond the current reference time.

a contact *start_time stop_time source_node dest_node xmit_data_rate*
The **add contact** command. This command schedules a period of data transmission from *source_node* to *dest_node*. The period of transmission will begin at *start_time* and end at *stop_time*, and the rate of data transmission will be *xmit_data_rate* bytes/second.

d contact *start_time source_node dest_node*
The **delete contact** command. This command deletes the scheduled period of data transmission from *source_node* to *dest_node* starting at *start_time*. To delete all contacts between some pair of nodes, use '*' as *start_time*.

i contact *start_time source_node dest_node*

This command will print information (the stop time and data rate) about the scheduled period of transmission from *source_node* to *dest_node* that starts at *start_time*.

l contact

This command lists all scheduled periods of data transmission.

a range *start_time stop_time one_node the_other_node distance*

The **add range** command. This command predicts a period of time during which the distance from *one_node* to *the_other_node* will be constant to within one light second. The period will begin at *start_time* and end at *stop_time*, and the distance between the nodes during that time will be *distance* light seconds.

d range *start_time one_node the_other_node*

The **delete range** command. This command deletes the predicted period of constant distance between *one_node* and *the_other_node* starting at *start_time*. To delete all ranges between some pair of nodes, use '*' as *start_time*.

i range *start_time one_node the_other_node*

This command will print information (the stop time and range) about the predicted period of constant distance between *one_node* and *the_other_node* that starts at *start_time*.

l range

This command lists all predicted periods of constant distance.

m utcdelta *local_time_sec_after_UTC*

This management command sets ION's understanding of the current difference between correct UTC time and the time values reported by the clock for the local ION node's computer. This delta is automatically applied to locally obtained time values whenever ION needs to know the current time. For machines that use UTC natively and are synchronized by NTP, the value of this delta should be 0, the default.

m clockerr *known_maximum_clock_error*

This management command sets ION's understanding of the accuracy of the scheduled start and stop times of planned contacts, in seconds. The default value is 1. When revising local data transmission and reception rates, *ionadmin* will adjust contact start and stop times by this interval to be sure not to send bundles that arrive before the neighbor expects data arrival or to discard bundles that arrive slightly before they were expected.

m clocksync [{ 1 | 0 }]

This management command reports whether or not the computer on which the local ION node is running has a synchronized clock, as discussed in the description of the *ionClockIsSynchronized()* function (*ion* (3)).

If a Boolean argument is provided when the command is executed, the characterization of the machine's clock is revised to conform with the asserted value. The default value is 1.

m production *planned_data_production_rate*

This management command sets ION's expectation of the mean rate of continuous data origination by local BP applications throughout the period of time over which congestion forecasts are computed. For nodes that function only as routers this variable will normally be zero. A value of –1, which is the default, indicates that the rate of local data production is unknown; in that case local data production is not considered in the computation of congestion forecasts.

m consumption *planned_data_consumption_rate*

This management command sets ION's expectation of the mean rate of continuous data delivery to local BP applications throughout the period of time over which congestion forecasts are computed. For nodes that function only as routers this variable will normally be zero. A value of –1, which is the default, indicates that the rate of local data consumption is unknown; in that case local data consumption is not considered in the computation of congestion forecasts.

m occupancy *heap_occupancy_limit* [*file_system_occupancy_limit*]
This management command sets the maximum number of megabytes of storage space in ION's SDR non-volatile heap, and/or in the local file system, that can be used for the storage of zero-copy objects. A value of –1 for either limit signifies "leave unchanged". The default heap limit is 60% of the SDR data store's total heap size. The default file system limit is 1 Terabyte.

m horizon { 0 | *end_time_for_congestion_forecasts* }
This management command sets the end time for computed congestion forecasts. Setting congestion forecast horizon to zero sets the congestion forecast end time to infinite time in the future: if there is any predicted net growth in bundle storage space occupancy at all, following the end of the last scheduled contact, then eventual congestion will be predicted. The default value is zero, i.e., no end time.

m alarm '*congestion_alarm_command*'
This management command establishes a command which will automatically be executed whenever *ionadmin* predicts that the node will become congested at some future time. By default, there is no alarm command.

m usage
This management command simply prints ION's current data store occupancy (the number of megabytes of space in the SDR non-volatile heap and file system that are occupied by bundles), the limit on occupancy, and the maximum level of occupancy predicted by the most recent *ionadmin* congestion forecast computation.

r '*command_text*'
The **run** command. This command will execute *command_text* as if it had been typed at a console prompt. It is used to, for example, run another administrative program.

s The **start** command. This command starts the *rfxclock* task on the local ION node.

x The **stop** command. This command stops the *rfxclock* task on the local ION node.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

EXAMPLES

@ 2008/10/05–11:30:00
Sets the reference time to 1130 (UTC) on 5 October 2008.

a range +1 2009/01/01–00:00:00 1 2 12
Predicts that the distance between nodes 1 and 2 (endpoint IDs ipn:1.0 and ipn:2.0) will remain constant at 12 light seconds over the interval that begins 1 second after the reference time and ends at the end of calendar year 2009.

a contact +60 +7260 1 2 10000
Schedules a period of transmission at 10,000 bytes/second from node 1 to node 2, starting 60 seconds after the reference time and ending exactly two hours (7200 seconds) after it starts.

SEE ALSO

ionadmin (1), *rfxclock* (1), *ion* (3)

NAME

ionsecrc – ION security policy management commands file

DESCRIPTION

ION security policy management commands are passed to **ionsecadmin** either in a file of text lines or interactively at **ionsecadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the ION security policy management commands are described below.

A parameter identifed as an *eid_expr* is an "endpoint ID expression." For all commands, whenever the last character of an endpoint ID expression is the wild-card character '*', an applicable endpoint ID "matches" this EID expression if all characters of the endpoint ID expression prior to the last one are equal to the corresponding characters of that endpoint ID. Otherwise an applicable endpoint ID "matches" the EID expression only when all characters of the EID and EID expression are identical.

COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }
Echo control. Setting echo to 1 causes all output printed by ionsecadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

1 The **initialize** command. Until this command is executed, the local ION node has no security policy database and most *ionsecadmin* commands will fail.

a key *key_name file_name*
The **add key** command. This command adds a named key value to the security policy database. The content of *file_name* is taken as the value of the key. Named keys can be referenced by other elements of the security policy database.

c key *key_name file_name*
The **change key** command. This command changes the value of the named key, obtaining the new key value from the content of *file_name*.

d key *key_name*
The **delete key** command. This command deletes the key identified by *name*.

i key *key_name*
This command will print information about the named key, i.e., the length of its current value.

l key
This command lists all keys in the security policy database.

a bspbabrule *sender_eid_expr receiver_eid_expr { " | ciphersuite_name key_name }*
The **add bspbabrule** command. This command adds a rule specifying the manner in which Bundle Authentication Block (BAB) validation will be applied to all bundles sent from any node whose endpoints' IDs match *sender_eid_expr* and received at any node whose endpoints' IDs match *receiver_eid_expr*. Both *sender_eid_expr* and *receiver_eid_expr* should terminate in wild-card characters, because both the security source and security destination of a BAB are actually nodes rather than individual endpoints.

If a zero-length string ('') is indicated instead of a *ciphersuite_name* then BAB validation is disabled for this sender/receiver EID expression pair: all bundles sent from nodes with matching administrative endpoint IDs to nodes with matching administrative endpoint IDs will be immediately deemed authentic. Otherwise, a bundle from a node with matching administrative endpoint ID to a node with matching administrative endpoint ID will only be deemed authentic if it contains a BAB computed via the ciphersuite named by *ciphersuite_name* using a key value that is identical to the current value of

the key named *key_name* in the local security policy database.

NOTE: if the security policy database contains no BAB rules at all, then BAB authentication is disabled; all bundles received from all neighboring nodes are considered authentic. Otherwise, BAB rules **must** be defined for all nodes from which bundles are to be received; all bundles received from any node for which no BAB rule is defined are considered inauthentic and are discarded.

c bspbabrule *sender_eid_expr receiver_eid_expr { " | ciphersuite_name key_name }*
The **change bspbabrule** command. This command changes the ciphersuite name and/or key name for the BAB rule pertaining to the sender/receiver EID expression pair identified by *sender_eid_expr* and *receiver_eid_expr*. Note that the *eid_expr*s must exactly match those of the rule that is to be modified, including any terminating wild-card character.

d bspbabrule *sender_eid_expr receiver_eid_expr*
The **delete bspbabrule** command. This command deletes the BAB rule pertaining to the sender/receiver EID expression pair identified by *sender_eid_expr* and *receiver_eid_expr*. Note that the *eid_expr*s must exactly match those of the rule that is to be deleted, including any terminating wild-card character.

i bspbabrule *sender_eid_expr receiver_eid_expr*
This command will print information (the ciphersuite and key names) about the BAB rule pertaining to *sender_eid_expr* and *receiver_eid_expr*.

l bspbabrule
This command lists all BAB rules in the security policy database.

a bsppibrule *sender_eid_expr receiver_eid_expr block type number { " | ciphersuite_name key_name }*
The **add bsppibrule** command. This command adds a rule specifying the manner in which Payload Integrity Block (PIB) validation will be applied to all bundles sent from any node whose administrative endpoint ID matches *sender_eid_expr* and received at any node whose administrative endpoint ID ID matches *receiver_eid_expr*.

If a zero-length string ('') is indicated instead of a *ciphersuite_name* then PIB validation is disabled for this sender/receiver EID expression pair: all bundles sent from nodes with matching administrative endpoint IDs to nodes with matching administrative endpoint IDs will be immediately deemed secure. Otherwise, a bundle from a node with matching administrative endpoint ID to a node with matching administrative endpoint ID will only be deemed secure if it contains a PIB computed via the ciphersuite named by *ciphersuite_name* using a key value that is identical to the current value of the key named *key_name* in the local security policy database.

c bsppibrule *sender_eid_expr receiver_eid_expr block type number { " | ciphersuite_name key_name }*
The **change bsppibrule** command. This command changes the ciphersuite name and/or key name for the PIB rule pertaining to the sender/receiver EID expression pair identified by *sender_eid_expr* and *receiver_eid_expr*. Note that the *eid_expr*s must exactly match those of the rule that is to be modified, including any terminating wild-card character.

d bsppibrule *sender_eid_expr receiver_eid_expr block type number*
The **delete bsppibrule** command. This command deletes the PIB rule pertaining to the sender/receiver EID expression pair identified by *sender_eid_expr* and *receiver_eid_expr*. Note that the *eid_expr*s must exactly match those of the rule that is to be deleted, including any terminating wild-card character.

i bsppibrule *sender_eid_expr receiver_eid_expr block type number*
This command will print information (the ciphersuite and key names) about the PIB rule pertaining to *sender_eid_expr* and *receiver_eid_expr*.

l bsppibrule
This command lists all PIB rules in the security policy database.

x *[{ ~ | sender_eid_expr } [{ ~ | receiver_eid_expr} [{ ~ | bab | pib | pcb | esb }]]]*
This command will clear all rules for the indicated type of security block between the indicated security source and security destination. If block type is omitted it defaults to ~ signifying ''all security blocks''. If both block type and security destination are omitted, security destination defaults to ~ signifying ''all security destinations''. If all three command-line parameters are omitted, then security source defaults to ~ signifying ''all security sources''.

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

EXAMPLES

a key BABKEY ./babkey.txt
Adds a new key named ''BABKEY'' whose value is the content of the file ''./babkey.txt''.

a bspbabrule ipn:19.* ipn:11.* HMAC_SHA1 BABKEY
Adds a BAB rule requiring that all bundles sent from node number 19 to node number 11 contain Bundle Authentication Blocks computed via the HMAC_SHA1 ciphersuite using a key value that is identical to the current value of the key named ''BABKEY'' in the local security policy database.

c bspbabrule ipn:19.* ipn:11.* ''
Changes the BAB rule pertaining to all bundles sent from node number 19 to node number 11. BAB checking is disabled; these bundles will be automatically deemed authentic.

SEE ALSO

ionsecadmin (1)

NAME

ltprc – Licklider Transmission Protocol management commands file

DESCRIPTION

LTP management commands are passed to **ltpadmin** either in a file of text lines or interactively at **ltpadmin**'s command prompt (:). Commands are interpreted line-by line, with exactly one command per line. The formats and effects of the LTP management commands are described below.

COMMANDS

? The **help** command. This will display a listing of the commands and their formats. It is the same as the **h** command.

Comment line. Lines beginning with **#** are not interpreted.

e { 1 | 0 }

Echo control. Setting echo to 1 causes all output printed by ltpadmin to be logged as well as sent to stdout. Setting echo to 0 disables this behavior.

v Version number. Prints out the version of ION currently installed. HINT: combine with **e 1** command to log the version number at startup.

1 *est_max_export_sessions*

The **initialize** command. Until this command is executed, LTP is not in operation on the local ION node and most *ltpadmin* commands will fail.

The command uses *est_max_export_sessions* to configure the hashtable 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 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.

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 *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*.

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.

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.

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

max_import_sessions is simply the neighoring engine's own value for the corresponding export session parameter; it is the neighboring engine's retransmission window size for this span. It reduces the chance of allocation of all available space in the ION node's data store to LTP reception sessions.

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* 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.

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

The **change span** command. This command sets the indicated span's configuration parameters to the values provided as arguments.

d span *peer_engine_nbr*

The **delete span** command. This command deletes the span identified by *peer_engine_nbr*. The command will fail if any outbound segments for this span are pending transmission or any inbound blocks from the peer engine are incomplete.

i span *peer_engine_nbr*

This command will print information (all configuration parameters) about the span identified by *peer_engine_nbr*.

l span

This command lists all declared LTP data interchange spans.

s '*LSI command*'

The **start** 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.

m screening { y | n }

The **manage screening** command. This command enables or disables the screening of received LTP segments per the periods of scheduled reception in the node's contact graph. By default, screening is disabled — that is, LTP segments from a given remote LTP engine (ION node) may be accepted even when they arrive during an interval when the contact graph says the data rate from that engine to the local LTP engine is zero. When screening is enabled, such segments are silently discarded. Note that when screening is enabled the ranges declared in the contact graph must be accurate and clocks must be synchronized; otherwise, segments will be arriving at times other than the scheduled contact intervals and will be discarded.

m ownqtime *own_queuing_latency*

The **manage own queuing time** command. This command sets the number of seconds of predicted additional latency attributable to processing delay within the local engine itself that should be included whenever LTP computes the nominal round-trip time for an exchange of data with any remote engine. The default value is 1.

x The **stop** command. This command stops all link service input and output tasks for the local LTP engine.

w { 0 | 1 | <activity_spec> }

The **LTP watch** command. This command enables and disables production of a continuous stream of user-selected LTP activity indication characters. A watch parameter of "1" selects all LTP activity indication characters; "0" de-selects all LTP activity indication characters; any other *activity_spec* such as "df{]" selects all activity indication characters in the string, de-selecting all others. LTP will print each selected activity indication character to **stdout** every time a processing event of the

associated type occurs:

d bundle appended to block for next session

e segment of block is queued for transmission

f block has been fully segmented for transmission

g segment popped from transmission queue

h positive ACK received for block, session ended

s segment received

t block has been fully received

@ negative ACK received for block, segments retransmitted

= unacknowledged checkpoint was retransmitted

+ unacknowledged report segment was retransmitted

{ export session canceled locally (by sender)

} import session canceled by remote sender

[import session canceled locally (by receiver)

] export session canceled by remote receiver

h The **help** command. This will display a listing of the commands and their formats. It is the same as the **?** command.

EXAMPLES

a span 19 20 5 1024 32768 2 'udplso node19.ohio.edu:5001'

Declares a data interchange span between the local LTP engine and the remote engine (ION node) numbered 19. There can be at most 20 concurrent sessions of export activity to this node. Conversely, node 19 can have at most 5 concurrent sessions of export activity to the local node. Maximum segment size for this span is set to 1024 bytes, aggregation size limit is 32768 bytes, aggregation time limit is 2 seconds, and the link service output task that is initiated when LTP is started on the local ION node will execute the *udplso* program as indicated.

m screening n

Disables strict enforcement of the contact schedule.

SEE ALSO

ltpadmin (1), *udplsi* (1), *udplso* (1)