N4C

Networking for Communications Challenged Communities:
Architecture, Test Beds and Innovative Alliances
Contract no: 223994

D2.2: Functional Specification for DTN Infrastructure Software

comprising

RFC 5050 Bundle Agent

and

Associated Components

Version 1.2

n4c-wp2-023-dtn-infrastructure-fs.doc

Folly Consulting and LTU
Elwyn Davies, Avri Doria
www.folly.org.uk, www.ltu.se

ABSTRACT

Starting in May 2008, N4C is a 36 month research project in the Seventh Framework Programme (www.cordis.lu/fp7). In cooperation between users in Swedish Lapland and Ko evje region in Slovenian mountain and partners, the project will design and experiment with an architecture, infrastructure and applications in field trials and build two test beds.

This document provides the functional specification of the Delay- and Disruption-Tolerant Networking (DTN) infrastructure software needed to support the testbeds to be developed in the N4C project and provide the networking communications for the applications to be developed and deployed during the project. It constitutes Deliverable 2.2 from Work Package 2 in N4C.

This functional specification essentially describes a subset of the functionality provided by the DTN Research Group (DTNRG) DTN2 reference implementation as at version 2.7.0 in early 2010, with some additional components that are essential to support test bed deployments that will be managed remotely and need to work autonomously over a longer period than may be usual for applications of the reference implementation. It is not a complete functional specification of DTN2 because DTN2 contains a number of components that are peripheral or irrelevant to the expected deployments in N4C testbeds.

The core functionality of the infrastructure implements a number of the experimental standards that have been developed by the DTN Research Group, especially The Bundle Protocol [RFC5050]. This functionality is implemented in the form of a user level server process designed to mediate the creation, maintenance, forwarding and delivery of DTN 'bundles' (as the unit of data transmission in DTN is known) between applications that use DTN communications and lower level communications capabilities on the machines where the infrastructure is deployed. The communications capabilities are expected to provide TCP and UDP over IP transport capabilities.

This version is preliminary and will be developed further in parallel with the software.

Due date of deliverable: 31/12/2009 Actual submission date: 30/04/2010

Document history			
Version	**Status**	**Date**	**Author**
1.2	Released version for N4C deliverable 2.2	30/04/2010	Avri Doria/E. Davies
1.1	Check of cross-refs	30/04/2010	Elwyn Davies
1.0	Revised version for review including MIBs and DING	28/04/2010	Avri Doria/E. Davies
0.9	Version for review.	28/04/2010	Elwyn Davies
0.8	Final pass, glossary, refs, proof read	13/04/2010	Elwyn Davies
0.7	Further creation. Forwarding log, commands, mgmt	13/04/2010	Elwyn Davies
0.6	Further creation. Storage	05/04/2010	Elwyn Davies
0.5	Further creation. Routers..	31/03/2010	Elwyn Davies
0.1 - 0.4	Further creation. Added all communications area.	30/03/2010	Elwyn Davies
0.0	Created	12/12/2009	Elwyn Davies

Dissemination level	
	Level
PU = Public	X
PP = Restricted to other programme participants (including the Commission Services).	
RE = Restricted to a group specified by the consortium (including the Commission Services).	
CO = Confidential, only for members of the consortium (including the Commission Services).	

CONTENT

1. INTRODUCTION 7
1.1 Reading This Document 7
1.2 What is Covered and What is Not in this Document 8
2. OVERVIEW AND REQUIREMENTS 11
2.1 DTN2 and N4C Infrastructure Functionality Compared 16
2.2 Numbers and Performance ExpectationS 17
2.2.1 Usage on Low End Machines 18
2.2.1.1 Application Connectivity 18
2.2.1.2 Data Throughput 18
2.2.1.3 Bundle Storage 18
2.2.1.4 User Community 19
2.2.1.5 Link Management 19
2.2.2 Usage on Medium Scale Server Machines 19
2.3 Existing System 19
2.4 Terminology 20
2.5 References 38
3. EXTERNAL INTERFACES OF BUNDLE DAEMON 41
4. FUNCTIONALITY 44
4.1 Logging 46
4.2 Textual Representations of Internal Structures 47
4.3 System Startup Controller 47
4.3.1 Detailed Functionality 47
4.4 DTN Application Interface server 48
4.4.1 Primary API (compatible with DTN2 at release 2.7.0) 49
4.4.1.1 Primary API Functions 50
4.4.1.1.1 dtn_open 50
4.4.1.1.2 dtn_close 51
4.4.1.1.3 dtn_errno 52
4.4.1.1.4 dtn_set_errno 52
4.4.1.1.5 dtn_build_local_eid 54
4.4.1.1.6 dtn_register 55
4.4.1.1.7 dtn_unregister 58
4.4.1.1.8 dtn_find_registration 58
4.4.1.1.9 dtn_change_registration 59
4.4.1.1.10 dtn_bind 60
4.4.1.1.11 dtn_unbind 62
4.4.1.1.12 dtn_send 62
4.4.1.1.13 dtn_cancel 65
4.4.1.1.14 dtn_recv 66
4.4.1.1.15 dtn_session_update 68
4.4.1.1.16 dtn_poll_fd 70
4.4.1.1.17 dtn_begin_poll 70
4.4.1.1.18 dtn_cancel_poll 71
4.4.2 Secondary API (based on D-Bus) 71
4.4.3 Registration Management Functionality 71
4.4.4 Signals Sent to Other Components 73
4.4.5 Functionality Invoked by Signals Sent to this Component 73
4.5 Bundle Daemon Core 74
4.5.1 Core Start-up and Shutdown 74
4.5.2 Core Functionality 76
4.5.3 Event Distribution and Event Handlers 77
4.5.4 Event Handlers 78
4.5.5 Additional Functionality Supporting the Event Handler 93
4.6 Bundle Factory and Fragmentation Manager 97
4.6.1 Bundle Structure 98
4.6.2 Bundle Construction and Dissection 100

4.6.2.1 Internal Representation 100
4.6.2.2 Bundle Structure Verification 101
4.6.2.3 Wire Format Conversion 103
4.6.2.4 Persistent Storage Serialization and Deserialization 105
4.6.2.5 API Conversion 106
4.6.3 Bundle Security Protocol Support 106
4.6.4 Bundle Fragmenter 108
4.6.5 Bundle Custody and Custody Timers 111
4.6.6 Forwarding Information and the Forwarding Log 112
4.7 Endpoint Identifier Management 113
4.7.1 Parsing Endpoint Identifiers 114
4.8 Bundle Router 115
4.8.1 Bundle Router Generic Functionality 115
4.8.2 Table-based Routing Model 117
4.8.2.1 Table Based Router Event Handlers 119
4.8.2.2 Table Based Router Supporting Functionality 121
4.8.3 Example Table Based Routers 123
4.8.3.1 Static Routing 123
4.8.3.2 Epidemic or Flood Routing 123
4.8.4 Dynamic Routing Manager 124
4.8.5 PRoPHET Routing 124
4.9 Contact Manager 126
4.9.1 Components 127
4.9.1.1 Convergence Layers 127
4.9.1.2 Types of Convergence Layer 128
4.9.1.2.1 Non-Connection Oriented Convergence Layers 128
4.9.1.2.2 Connection Oriented Convergence Layers 129
4.9.1.3 Interfaces 130
4.9.1.4 Links 130
4.9.1.5 Contacts 132
4.9.1.6 Discovery Agents 132
4.9.2 Interface Manager 133
4.9.3 Discovery Agent Manager 133
4.9.3.1 Discovery Manager Functionality 134
4.9.3.2 Generic Discovery Agent 135
4.9.3.3 Generic Announcement Functionality 136
4.9.3.4 Functionality for Specific Types of Discovery Agent 136
4.9.3.4.1 IP Discovery 136
4.9.3.4.2 Bonjour Discovery 138
4.9.3.4.3 Bluetooth Discovery 138
4.9.4 Link Management 138
4.9.4.1 Link Functionality Common to all Link Types 138
4.9.4.2 Functionality for Specific Link Types 142
4.9.4.2.1 Always On Link Specific Functionality 142
4.9.4.2.2 On Demand Link Specific Functionality 143
4.9.4.2.3 Opportunistic Link Specific Functionality 143
4.9.4.2.4 Scheduled Link Specific Functionality 143
4.9.5 Contacts 143
4.9.6 Convergence Layer Abstraction 144
4.9.6.1 Connection Oriented Convergence Layer Model 145
4.9.6.2 Stream Oriented Convergence Layer Model 148
4.9.7 Example Concrete Convergence Layers 155
4.9.7.1 TCP Convergence Layer 155
4.9.7.2 UDP Convergence Layer 157
4.9.7.3 Null Convergence Layer 160
4.9.8 Contact Manager 161
4.9.8.1 Link Set Management Functionality 162
4.9.8.2 Contact Manager Event Handler Routines 163
4.9.8.3 Creation of a New Opportunistic Link 164

4.9.8.4 Management of On Demand Link Availability 164
4.10 Storage Manager 164
4.10.1 Bundle Daemon Configuration and Global State....166
4.10.2 Bundle State 166
4.10.3 Registration State 166
4.10.4 Link State....166
4.10.5 PRoPHET Dynamic Routing State 166
4.11 Configuration and Management....166
4.11.1 Background 167
4.11.2 Management API 168
4.11.3 Management Interface....169
4.11.4 Configuration Change Notification 170
4.11.5 Logging and Log File Management 170
4.11.6 Management Information 170
4.11.6.1 Convergence Layer 170
4.11.6.2 Bundle Protocol....172
4.11.6.3 Routing....180
4.11.6.4 Info Plane 183
4.11.6.5 Application API....183
4.11.7 Configuration and Management Commands....183
4.11.7.1 API Server Control....184
4.11.7.2 Bundle Control....184
4.11.7.3 DTN2 Console Control 187
4.11.7.4 Discovery Control 187
4.11.7.5 Interface Control 189
4.11.7.6 Link Control....191
4.11.7.7 Log Control 197
4.11.7.8 Miscellaneous Commands 198
4.11.7.9 Bundle Daemon Parameter Control....198
4.11.7.10 Registrations Control....201
4.11.7.11 Routing Control....202
4.11.7.11.1 Static Router Control 203
4.11.7.12 External Router Controls....205
4.11.7.12.1 PRoPHET Routing Control....206
4.11.7.12.2 DTLSR Routing Control....208
4.11.7.13 Security Control 209
4.11.7.14 Shutdown Control....210
4.11.7.15 Storage Control....210
4.11.7.15.1 Configuration Settings for File System Database....211
4.11.7.15.2 Configuration Settings for Berkeley Database....211
4.11.7.15.3 Configuration Settings for External Persistent Storage Interface 213
4.12 Internal Applications 214
4.12.1 DTN Ping Echo Application....214
4.12.2 DTN Traceroute Application....214
4.12.3 DTN Management Interface Application....214
4.12.4 DTN Heartbeat Application....215
4.12.5 Administrative Bundle Handler 215
4.12.6 Delay Tolerant Link State Router (DTLSR) Application 215
4.12.7 External Router Application....215

1. INTRODUCTION

This document provides the functional specification of the Delay- and Disruption-Tolerant Networking (DTN) infrastructure software needed to support the testbeds to be developed in the N4C project and to provide the networking communications for the applications to be developed and deployed during the project. The architecture of this system is described in the *N4C System Architecture* [N4Carch]. For more information about DTN as a whole please refer to the *DTN- The State of the Art* document produced by the N4C project [DTNstateArt].

This functional specification essentially describes a subset of the functionality provided by the DTN Research Group DTN2 reference implementation as at version 2.7.0 in early 2010 [DTN2], with some additional components that are essential to support test bed deployments that will be managed remotely and need to work autonomously over a longer period than may be usual for applications of the reference implementation. It is not a complete functional specification of DTN2 because DTN2 contains a number of components that are peripheral or irrelevant to the expected deployments in N4C testbeds. In the future this functional specification may be expanded to cover the whole of DTN2.

The core functionality of the infrastructure implements a number of the experimental standards that have been developed by the Internet Research Task Force (IRTF) DTN Research Group (DTNRG) [DTNRG], especially The Bundle Protocol [RFC 5050]. This functionality is implemented in the form of a user level server process designed to mediate the creation, maintenance, forwarding and delivery of DTN ‘bundles’ (as the unit of data transmission in DTN is known) between applications that use DTN communications directly and the lower level communications capabilities on the machines where the infrastructure is deployed. The communications capabilities are expected to provide at least TCP and UDP over IP transport capabilities.

The documents published by the DTNRG as RFCs and Internet Drafts define many parts of the required functionality. Where appropriate the functionality of this infrastructure software will be defined by reference to these documents.

The functionality as defined in this document will, wherever possible, be defined and modularized in such a way that it is extensible. The way in which it is modularized will be guided by the experience gained in designing and building the DTN2 implementation. This will facilitate extending the specification in future both to cover the totality of the existing DTN2 functionality for the purposes of documentation, and also to allow new functionality to be added - such as alternative routing protocols and extensions to the types of blocks carried within the Bundle Protocol framework.

1.1 READING THIS DOCUMENT

This Functional Specification is a long and complex document. It contains descriptions of the components that make up the Bundle Daemon (BD) and the functions they have to implement both at an overview level and in detail through a large number of sequence numbered paragraphs. These numbered items are referred to as *item <n>* in the text and can be used to trace the functionality implemented back to this specification. These descriptions start with *item 1* in Section 4.1 and run on to *item 513* at the end of Section 4.

To obtain an overview of the functionality of the Bundle Daemon, it is recommended that you read Sections 1 to 3 and then the preambles of the sub-sections of Section 4 which contain the overviews of the various components, omitting the sequence numbered detailed functionality sections on a first reading.

There is an extensive glossary of terminology and abbreviations in Section 2.4, which covers both generic DTN terms and a selection of key items that feature in many parts of this document.

1.2 WHAT IS COVERED AND WHAT IS NOT IN THIS DOCUMENT

As mentioned in Section 1, this document is the specification for the BD that is needed for the N4C infrastructure. It is to a large extent 'reverse engineered' from the existing DTN2 implementation, but this document focuses on the functionality that is of prime importance for the N4C project deployment. Thus it is not a complete functional specification of DTN2 as currently implemented and there are a small number of items that do not yet exist in DTN2. There are two significant consequences:

- There are some problems and errors in the existing DTN2 code. There are a number of notes in this document enclosed in square brackets [...] that note areas where DTN2 either diverges from the functionality needed by N4C or is potentially broken.
- The reader will find that there are a number of pieces of functionality that are mentioned but not described in detail.

One area of functionality affected by this is the Publish/Subscribe mechanism. There are pieces of functionality that support this mechanism scattered throughout the BD, so it is not easy to totally eliminate it from descriptions. This functionality is omitted because N4C does not currently have a clear use case for the functionality although it could possibly be useful in areas such as Network Management, and also because it does not yet have a published specification as an Internet Draft or Request For Comment (RFC). More information about the Publish/Subscribe mechanism can be found in the paper *The Design and Implementation of a Session Layer for Delay Tolerant Networks* [PubSub].

A number of other more localized pieces of functionality are also not described in detail:

- **External Router Interface**: This interface is designed to allow router functionality (as described in Section 4.8) to be implemented as a separate process that communicates with the main BD through an XML-encoded inter-process message passing protocol. Detailed description of this interface and how it could be used is omitted because the XML-encoded interface and the separate process implementation are probably inappropriate for a production environment and would require significant extra resources from the platform that would be inappropriate for the sort of low end platforms that are likely to be used in N4C. Also there are no open source router implementations that use the External Router Interface at present. The DTN2 code documentation area contains a document (*doc/external-router-interface.html*) that outlines the functionality of the interface. More details can be found in the documentation of the *plugin-architecture* in the same location.
- **External Convergence Layer**: This interface is designed to allow extra convergence layers to be provided in external processes communicating through an

XML-encoded message passing interface similar to the External Router Interface just described. This interface is only described in the *plugin-architecture*. As with the External Router Interface, this does not seem to be an appropriate subsystem for N4C applications and there are no freely available convergence layers that use the interface.

- **Delay Tolerant Link State Router (DTLSR)**: This router mechanism extends the link state routing protocols used in conventional IP networks into a form that is usable in DTN networks. It is applicable to situations where there is reasonably stable underlying topology of communication links, but some of these links may be intermittently unavailable. Situations such as communications with data mules travelling on predetermined routes between static nodes is an example scenario where it might be useful. The protocol is documented in a paper [DTLSR], but there is no RFC or Internet Draft specifying the router or protocols as yet.
- **TCA (Tetherless Computing Architecture) Router**: For more details see [TCA]. Instructions for setting up and running this router can be found at [TCAhowTo].
- **Bluetooth RFCOMM Convergence Layer**: Operates similarly to the TCP/IP Convergence Layer (see Section 4.9.7.1) using the Bluetooth Radio Frequency Communication (RFCOMM) transport [RFCOMM].
- **Ethernet Frame Convergence Layer**: A simple non-connection oriented convergence layer that uses Ethernet MAC addresses and carries bundles in Ethernet frames, one frame to one bundle with similar limitations to the UDP convergence layer (see Section 4.9.7.2). [This convergence layer was originally intended to be used in conjunction with the NeighborhoodRouter (a very simple router for local networks). The actual router was deleted from DTN2 on 2007/11/07 - Mercurial changeset 3002:251376020e19 - but there are comments relating to it in the *dtn.conf* file and the Ethernet convergence layer. DTLSR can possibly be used instead.]
- **File Convergence Layer**: In theory this convergence layer passes bundles as opaque data 'blobs' in platform files. In practice it is not implemented in the current version of DTN2.
- **NORM (NACK Oriented Reliable Multicast over IP) Convergence Layer**: Described in the file *doc/norm_conv_layer.txt* supplied with the DTN2 code.
- **LTP (Licklider Transmission Protocol over UDP over IP) Convergence Layer**: This Convergence layer is under development and will be arriving soon [RFC5325]. It may be used in some parts of the N4C deployment but will be discussed in documents relating to those developments.
- **Serial Convergence Layer**: Connection oriented convergence layer operating over a point-to-point serial link (such as an RS.232 channel).

The set of components covered by this functional specification are summarized in the introduction to Section 4. Two sections may be of particular interest to DTN application writers and users of the BD:

- Section 4.4.1 contains a detailed description of the existing DTN2 Application Programming Interface that may be useful for application writers.

- Section 4.11 contains a complete list of the configuration and management commands together with their parameters as implemented in DTN2 at version 2.7.0. Unlike the remainder of the document this section attempts to give a complete view of the commands including those that apply to components that are not expected to

be used in N4C. The intention is that this should be extracted and used separately as part of the manual for DTN2.

2. OVERVIEW AND REQUIREMENTS

The core functionality of the N4C infrastructure is implemented by the Bundle Daemon which is a software server process running in a single machine. The Bundle Daemon (BD) implements the functionality needed to manage the sending and receiving of DTN 'bundles' over network communication links as specified in RFC 5050 [RFC5050] and associated documents, and described as the 'bundle (protocol) agent' (BPA) in RFC 5050.

The core 'philosophy' of a DTN network is to provide a *Store, Carry and Forward* service for bundles that is very close to the email paradigm. In comparison to the core philosophy of conventional IP networks, where intermediate nodes and endpoint communication daemons do not offer more than very transient storage of packets in transit, a DTN BD has to be prepared to store bundles for significant periods until an appropriate forwarding opportunity or delivery opportunity occurs, or the bundle's lifetime expires. To match the network philosophy, the BD at the destination node where a bundle should be delivered has to offer the capability for deferred delivery. To implement this, an application can install state in the BD in the form of a *registration* of a destination address. Depending on the specification of the registration, the BD may retain bundles for later (deferred) delivery even if the application that made the registration is not active when the bundles arrive. Thus in contrast to conventional IP usage, bundles will not necessarily be dropped when they arrive at their destination and there is no application waiting to receive them immediately.

According to RFC5 5050, a BPA must offer the following services to applications:

- commencing a registration (i.e., indicating that this endpoint will take delivery of suitably addressed bundles);
- terminating a registration;
- switching a registration between Active and Passive states;
- transmitting a bundle to an identified bundle endpoint;
- canceling a transmission;
- polling a registration that is in the passive state;
- delivering a received bundle.

The nature of these operations is described in more detail in RFC 5050 and greatly expanded in this document.

The BD also handles

- configuration and management of the BD, including remote management from other nodes,
- the encapsulation of application data for transmission on behalf of applications,
- delivery of bundles to applications in accordance with the destination node locators (known as Endpoint Identifiers or EIDs) together with any demultiplexing information registered by the application (using *service tags)*,
- management of the resources of the node that are made available for DTN operations including the temporary and permanent storage for bundles needed to implement the 'store, carry and forward' paradigm of DTN,
- discovery of and management of links to other DTN capable nodes providing connections over which this node is able to exchange bundles with its peers,

- management of the routing information needed to control the forwarding of bundles on the connections established to DTN capable peers including exchanging routing information needed for dynamic routing protocols, and
- forwarding of bundles to connected DTN capable peers in accordance with the routing information appropriate for that connection and seeking to make optimum usage of the resources of this node.

Links to other DTN capable nodes used in N4C will be expected to use TCP and/or UDP over IP for communication but other transport protocols and link layers could also be used. Hence for the purposes of the physical media used for the links may be of any type so long as they at least support IP communications, but it is expected that many of them will use wireless connectivity such as IEEE 802.11 (Wi-Fi). This specification does not at present consider in detail other types of connection or transport protocol although a number of other communication modes are supported by the DTN2 reference implementation.

The implementation for N4C will be primarily concerned with opportunistic connections over low latency links where bidirectional communications are realistic rather than for long latency links where unidirectional communications are the norm. As such it should be optimized for the opportunistic case where appropriate.

The BD server process has interfaces to:

- Lower level communications facilities including TCP and UDP over IP
- Applications requiring to send and/or receive bundles
- Management and configuration mechanisms for the server and communications
- Persistent storage to maintain bundles when the server is not running actively

The BD provides an extensible routing capability, which will allow a number of different routing functions to be provided by the daemon. Multiple different routing functions may be active at once, selected according to the connection and negotiation at the time of connection. The various routing protocols will be provided by means of dynamically loadable plug-ins that can be loaded at start up or during operation of the server. The initial implementation of the server will provide at least:

- Static table-based routing
- Epidemic-style flood routing
- PRoPHET delivery probability- based routing

The BD provides functionality to support generic information exchange needed by dynamic routing protocols that negotiate over the identity of the bundles to be exchanged during opportunistic encounters, such that this can be used by several different routing protocols. This type of dynamic routing protocol is only viable if a low latency reliable link can be established locally between nodes involved in an opportunistic encounter. The local information exchange will therefore take place using a separate TCP-based connection rather than being encapsulated in bundles. It is possible that additional information could be exchanged in bundles, possibly over multiple hops, to assist the routing system.

The BD provides an extensible bundle convergence layer interface supporting various underlying communication protocols. This is provided through dynamically loadable plug-ins [DTN2 in its current form offers extensibility either at compile time or through an XML-based inter-process message passing interface linking to an *external router*.] These plug-ins can be loaded either at start-up or in response to configuration of additional links that require the convergence layer. The daemon supports at least

- TCP/IP Stream Convergence layer
- UDP/IP Convergence layer
- Null Convergence Layer

[DTN2 supports a number of other convergence layers that are not expected to be useful in N4C deployments, including layers that use Bluetooth RFCOMM, NORM IP Multicast, Raw Ethernet frames, serial links such as RS232, file based storage, and a general purpose external convergence layer interface driven over an XML-encoded message passing inter-process communications link. A convergence layer using the Licklider Transmission Protocol running over UDP/IP is under development and is expected to be incorporated into DTN2 shortly. See [RFC5325], [RFC5326] and [RFC5327]. See Section 1.2 for more details.]

The TCP/IP stream convergence layer will be written in such a way that the common functionality needed for alternative stream protocols can be factored out and provided as a separate module.

The BD supports an extensible neighbour discovery mechanism. This will be implemented through loadable plug-ins that can be dynamically loaded either at start up or on demand if needed by a new link (type). The daemon will support at least:

- IP based discovery as implemented by DTN2.
- Bonjour-based discovery.

The BD supports persistent storage of bundles and forwarding state associated with such bundles so that the bundles can be maintained and delivered across a period when the daemon is shut down and subsequently reactivated. [Note: this requires some improvements over the current DTN2 implementation that appears to not maintain the forwarding state satisfactorily because there is no persistency associated with link specifications. This needs to be improved.]

The BD supports a basic suite of security mechanisms taken from the Bundle Security Protocol to support authentication and integrity protection for bundles. The BD is implemented in such a way that additional security suites can be added as they become necessary and the relevant security algorithms are implemented.

The BD supports management and configuration through a protocol carried over bundles using the Diagnostic Interplanetary gateway protocol (DING) [Ding]. This protocol carries management information using a specially designed Management Information Base (MIB) compatible with SNMP MIBs as used in the conventional Internet, and hence accessible through existing management tools via a suitable gateway. This document also specifies the set of low level management commands that are used to configure the existing DTN2 implementation.

The BD implements the functionality required to support the protocol components defined in the following RFCs and Internet Drafts:

Defining Document	Version	Document Title	Functionality Implemented
RFC 5050 [RFC5050]	N/A	Bundle Protocol Specification	• Provision of the application services defined in Section 3.3 • Creation and interpretation of the complete bundle format defined in Section 4 including support of blocks defined in extensions. • Creation, transmission, reception and processing of all types of administrative record specified in Section 5.1 and Section 6. • Custody management as specified in Sections 5.10 - 5.12 and 6.3. • Bundle processing as specified in Sections 5.2 - 5.13. • Interface to the convergence layer as specified in Section 7.

Defining Document	Version	Document Title	Functionality Implemented
draft-irtf-dtnrg-bundle-security [BSP]	15	Bundle Security Protocol Specification	• Creation and interpretation of the security blocks defined in Sections 2.1 - 2.5. • For this version, it is assumed that key and certificate management will be performed out of band by the management system, typically using pre-shared keys. • Management of security policy configuration and retrieval as required by Section 3.1. • Bundle canonicalization procedures as defined in Section 3.4. • Reactive fragmentation will not be implemented for secured bundles (Section 3.9). • The mandatory cipher suites will be implemented as defined in Section 4.
draft-irtf-dtnrg-bundle-metadata-block [MetadataBlock]	07	Delay-Tolerant Networking Metadata Extension Block	• The implementation will support metadata blocks of the predefined types. • Additional metadata block types may be defined to support the extensions of the dtn: URI scheme [TBD].
draft-irtf-dtnrg-bundle-previous-hop-block [PrevHopBlock]	11	Delay-Tolerant Networking Previous Hop Insertion Block	• The insertion and deletion of Previous Hop Insertion Blocks will be supported.
draft-irtf-dtnrg-dtn-uri-scheme [URIscheme] + draft-davies-dtnrg-uri-find [URIfind]	00 01	The DTN URI Scheme	• This functionality is still under development and will require further investigation. • The ability for nodes to have multiple EIDs will be supported. • The ability of some of these EIDs to be ‘service names’ will be supported.

Defining Document	Version	Document Title	Functionality Implemented
draft-irtf-dtnrg-prophet [PRoPHET]	05	Probabilistic Routing Protocol for Intermittently Connected Networks	• Full functionality of PRoPHET will be supported. • Routing metadata will be exchanged over a reliable (TCP) channel separated from the bundle exchange channel. • The forwarding strategies in Section 3.6 will be implemented. • The queuing strategies in Section 3.7 will be implemented.
draft-irtf-dtnrg-tcp-clayer [TCPclayer]	02	Delay Tolerant Networking TCP Convergence Layer Protocol	• The basic functionality of the TCP Convergence Layer is implemented as defined in Sections 4, 5.2 and 6. • The additional optional functionality as defined in Section 5.2-5.5 is implemented.
draft-irtf-dtnrg-udp-clayer [UDPclayer]	00	UDP Convergence Layers for the DTN Bundle and LTP Protocols	• This is a very basic specification of how bundles can be encapsulated in a UDP datagram.
draft-irtf-dtnrg-ding-network-management [Ding]	02	DING Protocol -- A Protocol For Network Management	• Protocol for exchange of network management and configuration information over bundles using Management Information Based scheme.
dtn.mib [DTNmib]	-	dtnMIB MODULE	• Draft MIB for DTN.

The BD provides a built-in DTNping echo capability to respond to DTN pings and a number of other internal applications to support management and routing.

2.1 DTN2 AND N4C INFRASTRUCTURE FUNCTIONALITY COMPARED

Figure 1 shows how the functionality of DTN2 and the functionality used for the N4C infrastructure overlaps, and the functionality that is not common to the two pieces of software. Apart from the changes to the PRoPHET routing protocol, the additional items in the N4C implementation do not prevent interoperation of the software.

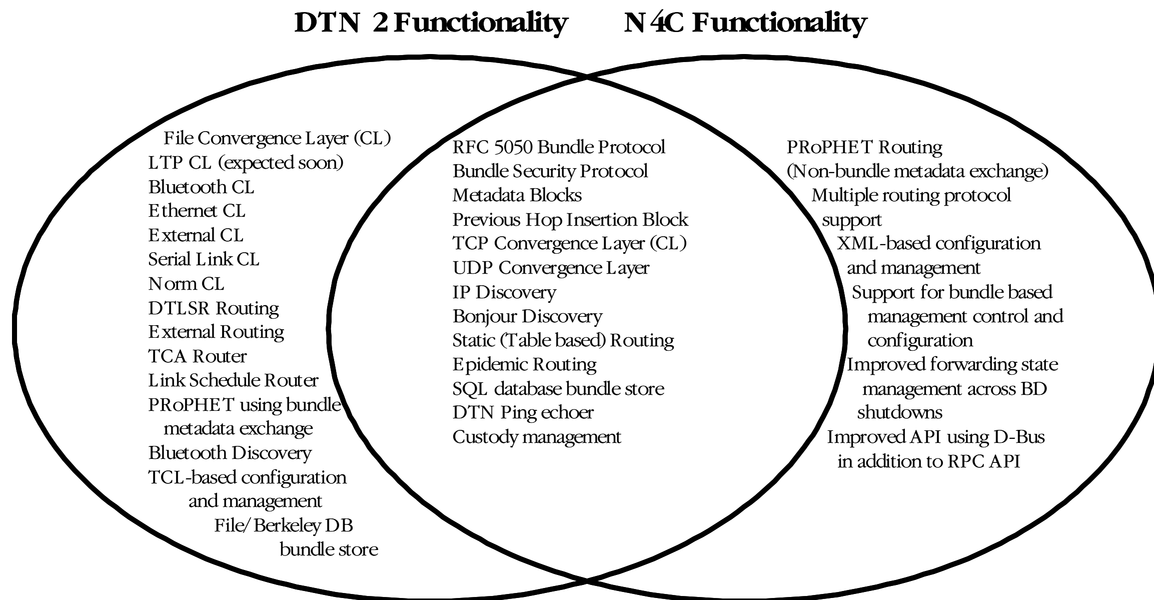

FIGURE 1 DTN2 AND N4C INFRASTRUCTURE FUNCTIONALITY OVERLAP

As of version 2.7.0 of DTN2, the PRoPHET implementation in DTN2 is not compliant with the version that is being proposed for publication as an experimental standard and hence, will be deprecated in future versions. It is intended that the DTN2 PRoPHET implementation should be updated to conform to the proposed standard and it would then interoperate with the N4C implementation.

Also in version 2.7.0 of DTN2, the implementation of the Bundle Security Protocol is out of date with respect to the version that is proposed for publication as an experimental standard (it corresponds approximately to version 05 of the Bundle Security Protocol draft as compared with the latest version 15). Work is currently ongoing outside N4C to update the DTN2 implementation to match it to the specification version that is to be standardized.

2.2 NUMBERS AND PERFORMANCE EXPECTATIONS

The DTN Infrastructure Bundle Daemon is expected to be deployed on a number of different types of system. These systems are generally expected to range from very compact low performance systems such as sensor devices through various types of handheld systems such as ‘Internet Tablets’ (e.g., Nokia N810/N900) and ‘Smart Phones’ (e.g., Android phones, Apple iPhone, and Nokia E61), netbooks (e.g., Asus EEE PC range), wireless router controllers up to medium scale desktop computers. Although it may be that this software will be deployed on more powerful machines, the nature of DTN networks makes it unlikely that the software will have to deal with very high throughput rates or handle high capacity network interfaces at full rate in the N4C project. Thus the design and implementation of this functionality should concentrate on compact and efficient implementation with maximum efficiency in use of processing capability. In particular, on low end machines, the software can expect to be idle for a large proportion of the time. Deployments of this software will be concentrated in communications challenged areas, which will frequently also exhibit power availability challenges. The

implementation of the functionality should therefore aim to allow processors to enter as deep a sleep state as is possible to conserve power during idle periods.

2.2.1 Usage on Low End Machines

The intention of the N4C project is to demonstrate how applications running in a DTN environment can seamlessly integrate with applications running in the 'traditional' Internet. Some low end machines may well be designed to provide an interface to sensors, in which role they will be static and in most cases they will serve primarily as a source of bundles containing information retrieved from the attached sensors. However we expect that many other low end machines will be mobile devices carried around by human (and possibly animal) users. In the absence of more conventional networking infrastructure, N4C expects these users *to be the network*, i.e., their machines will not only act as interfaces to the human user carrying the machine, but will also function as *data mules* accepting and carrying bundles for other destinations and looking to offload them to other machines that may be encountered opportunistically and indicate that they can provide a route to deliver the bundle toward its destination.

2.2.1.1 Application Connectivity

On sensors and handheld systems the DTN infrastructure BD software will almost always be servicing the equivalent of a single user. The BD will be expected to support in the range of one to 10 applications in parallel but typically only a few of these will be actively sending or receiving data simultaneously because of the limitations of the human user and the small scale of the machines. The BD must be able to manage different, and possibly multiple, EIDs for each application.

2.2.1.2 Data Throughput

The throughput of the BD must be adequate to provide responsive performance for the human user, but the amount of data to be processed will typically be limited by the availability of connections to other machines. In an opportunistic encounter DTN environment, the useful performance of the BD will typically be constrained by the amount of data that can be transmitted and/or received over the wireless connection during short periods of connectivity. The most critical performance requirement is that the BD should be able to make effective use of the available connectivity. This requires that the communications overhead is kept as low as possible and that the BD is able to saturate the link bandwidth of typical wireless links connected to handheld devices (generally IEEE 802.11 Wi-Fi).

2.2.1.3 Bundle Storage

In a single user environment, the BD should be capable of managing up a few thousand (say, 5000) bundles. The bundle storage may be used not only for bundles that have been locally created or are awaiting local delivery, but must also support the *data* mule role of the machine. The BD must not place an inherent limit on the amount of bundle storage used, either per bundle or in total, although the capabilities of the device and the operating system may effectively impose limits, and user configuration may impose a quota on the bundle storage.

2.2.1.4 User Community

Although a handheld device will frequently be used by just a single human throughout its life, the BD must not constrain usage to a single user.

2.2.1.5 Link Management

The BD should be able to manage at least five simultaneous links (subject to operating system constraints, if any) in case the machine encounters several other machines at one time.

2.2.2 Usage on Medium Scale Server Machines

The major difference between use on handheld devices and on server machines is that many of the applications accessing the BD services will be automated processes with no direct human intervention. They will, therefore, typically be capable of generating and receiving a greater throughput of bundles. There may also be more applications. However although the overall size of the DTN network and machine community is unlimited, it is expected that the community will be scaled by using 'associations' of users that have some common characteristic (e.g., geographic region connection or specific interest). Such associations are expected to have memberships in the range one individual to a few hundred.

2.3 EXISTING SYSTEM

There are a number of existing implementations of the Bundle Daemon functionality in existence. In N4C, two of these are of primary importance:

- **DTN2 reference implementation**
  As of version 2.7.0, the DTN2 implementation is essentially compliant with the base DTN Bundle Protocol specification [RFC5050] but the implementation of the Bundle Security Protocol is considerably out of date., being based on an earlier version of the draft [BSP]. Work is in progress to update DTN2 to conform to the version of [BSP] which is expected to become an experimental standard during 2010.
- **SNC PRoPHET implementation**
  This implementation uses the most up to date version of the PRoPHET specification but the bundles exchanged are based on a much earlier version of the basic specification so that it will not interoperate with RFC 5050 implementations such as DTN2, and the implementation is incomplete in various other ways. However, it uses the Qt programming framework which improves its portability. See [SNCPRoPHET].

Additionally an Android implementation developed at KTH for the Bytewalla projects has been made available to the N4C project [Bytewalla].

2.4 TERMINOLOGY

In the explanations of terms in this section words in *italics* would have their own explanation elsewhere in the section.

Term	Explanation	More details
Action Type	Attribute of *Routes*. Either FORWARD_ACTION where a *Bundle* is forwarded to just one *Next-Hop* or COPY_ACTION where it is forwarded along every applicable route.	**4.8.1**
Active Registration	*Registration* to which *bundles* can currently be delivered.	**4.4**
Administrative Bundle	*Bundle* with the *Administrative Record* flag set in the *Bundle Control Flags*. May be a *Custody Signal* or a *Status Record*.	**4.12.5**
Administrative EID	*EID* to which *Custody Signals* are sent. In DTN2 this is the *Local EID* with an empty *Service Tag*.	**4.12.5**
Advertisement	Synonym for *Announcement*.	**4.9.1.6, 4.9.3.3**
All Bundles List	In the *Bundle Core*, a list of all *Bundles* currently extant in the BD.	**4.5.1, 4.6.2**
All Bundles List	In the *Flood* router, a list of all *Bundles* that will be sent to any new *Next-Hop* that is discovered.	**4.8.3.2**
Always On	Type of *Link* that has a *Contact* open at all times, if the underlying transport can provide it.	**4.9.1.4**
Announcement	Message sent by a *Discovery Agent* to advertise that a BD will accept communications using a specified *Convergence Layer* at a specified transport address.	**4.9.1.6,**
API	Application Programming Interface	**4.4**
API Server	Component that implements the *API* in the *BD*.	**4.4**
Application Programming Interface	Set of routines available over an RPC interface that allows applications to use the capabilities of the *BD*.	**4.4**
BAB	Bundle Authentication Block	[BSP], **4.6.1**
BD	Bundle Daemon.	

Term	Explanation	More details
Block	*Bundles* are constructed from a sequence of a *Primary* Block and one or more other *Blocks*. *Blocks* have a *Canonical Block Header* format that identifies the block type.	**4.6**
Block Control Flags	Set of flags present in each *Block* of a *Bundle* other than the *Primary Block*.	**[RFC5050], 4.6, 4.6.2.2**
Bluetooth	Personal Area Networking communication technology.	
BP	*Bundle Protocol*.	**[RFC5050]**
BPA	*Bundle Protocol Agent*.	**[RFC5050]**
BSP	*Bundle Security Protocol*.	**[BSP], 4.6.3**
Bundle	Basic unit of transmission of data in RFC 5050 *DTNs*.	**[RFC5050],**
Bundle Authentication Block	BAB is used to assure the authenticity and integrity of the bundle along a single hop from forwarder to (possibly) intermediate receiver.	[BSP]**, 4.6.1**
Bundle Control Flags	Set of flags in the *Primary Block* of a *Bundle*.	**[RFC5050], 4.6, 4.6.2.2**
Bundle Core	Core component of *DTN2* infrastructure software providing the main event processing loop, event processing routines and distribution of events to other components. Implements the *BPA* functionality and integrates it with the *Bundle Factory*, the *Communication Components,* the *API Server*, the *Bundle Routers* and the *Management Component*.	**4.5**
Bundle Daemon	An instance of this software.	
Bundle Endpoint	See *Endpoint*.	**4.7**
Bundle Factory	Component of the *BD* that manages the creation, destruction and transformation between formats of *Bundles*.	**4.6**
Bundle Fragment	Also *Fragment*. See *Fragmentation*.	**4.6.4**
Bundle ID	Unique number identifying a *Bundle*. Preserved across *BD* shutdown and restart until the	**4.6.2**

Term	Explanation	More details
	Persistent Storage database is reinitialized.	
Bundle Node	Platform supporting a *Bundle Daemon* and generally *DTN* applications.	
Bundle Protocol	Fundamental protocol used to transfer *Bundle* data units in a *DTN* as endorsed by the *IRTF DTN Research Group*. See **[RFC5050]**.	**[RFC5050]**
Bundle Protocol Agent	The bundle protocol agent of a node is the node component that offers the BP services and executes the procedures of the *Bundle Protocol* as defined in RFC 5050**.**	**[RFC5050]**
Bundle Reassembly	See *Fragmentation*.	**4.6.4**
Bundle Router	Component that manages *Routes* to other *BD* supporting nodes and determines the *Next-Hop* to which eligible *Bundles* should be *forwarded*.	**4.8**
Bundle Security Protocol	Specification of additional protocol capabilities that provide data integrity and confidentiality services for the *Bundle Protocol*.	**[BSP], 4.6.3**
Canonical Block Header	Format for the header of all *Blocks* in a *Bundle* as specified in **[RFC5050]**. Allows the *BD* to process types of *Blocks* that it does not explicitly understand as opaque data.	**4.6.2.2**
CB	*Confidentiality Block*.	[BSP]**, 4.6.1**
Chain Length Limit	Constraint on the (Table-based) *Bundle Router*. Limits the number of *Waypoint Routes* that should be used to determine the appropriate *Next-Hop* that should be used to *Forward* the bundle.	**4.8.2.2**
CL	*Convergence Layer*.	**4.9.1.1, 4.9.6**
Communication Components	The collection of components that provide the interface between the *BPA* and the data transport network in the *BD*. Includes *Convergence Layers*, *Links*, *Contacts*, *Contact Manager*, *Discovery Agents*, and *Announcements*.	**4.9**
Confidentiality Block	Previous name for *Payload Confidentiality Block*.	[BSP]**, 4.6.1**
Connection Message Queue	Queue of messages used to communicate between the main *Bundle Core* thread and the subsidiary connection control thread in a *Connection Oriented Convergence layer*.	**4.9.6.1**

Term	Explanation	More details
Connection Oriented CL	*CL* in which the underlying transport protocol offers a connection-based service, usually offering a reliable transport, possibly with stream based semantics and guaranteed in-order delivery of data.	**4.9.1.2.2,**
Contact	Abstraction of an active communication *Link* to a *next-hop* node. When a *Link* has an active *Contact* the link is said to be *open* and the *Contact* is *up*. Conversely the *Link* will be *closed* when the *Contact* is deactivated or goes *down*.	**4.9.1.5, 4.9.5**
Contact Header	Initial message exchanged during the opening of a *Contact* on a *Connection Oriented CL*.	**4.9.6.2, [TCPclayer]**
Contact Manager	Central component of the *Communications Component* that manages the *Links* and *Contacts* as they are created, change state and are destroyed.	**4.9.8**
Convergence Layer	Component that implements the functionality required to mediate between the *BPA* and an underlying transport network using a specified transport protocol and type of communication endpoint.	**4.9.1.1, 4.9.6**
COPY_ACTION	See *Action Type*.	**4.8.1**
Custodian	Node with *BD* that has agreed to accept *Custody* of a *Bundle*.	**4.6.5, 4.12.5**
Custodian EID	Field in a *Bundle* containing the *Administrative EID* of the node that currently claims *Custody* of the bundle.	**4.6.5, 4.12.5**
Custody	Contract offered with respect to a *Bundle* by a node implementing a *DTN BD*. If a node accepts *Custody* of a *Bundle*, it contracts to maintain the node in reliable, persistent storage for the *Bundle's* specified lifetime and to seek to forward the *Bundle* either to another node that will accept *Custody* or to its final destination *EID*. In effect a node accepting *Custody* becomes a surrogate for the original source node.	**4.6.5, 4.12.5**
Custody Bundles List	List maintained by the *Bundle Daemon Core* referencing all *Bundles* for which this node	**4.5.1, 4.6.5**

Term	Explanation	More details
	has accepted *Custody.*	
Custody Signal	Message sent by a node accepting *Custody* of a bundle to signal to the previous *Custodian* to indicate that it has accepted custody of a *Bundle* or is notifying various other events related to *Custody*. See Section 6.1.2 of **[RFC5050]**.	**4.6.5, 4.12.5**
Custody Timer	A timer is started whenever a *Bundle* for which this *BD* has accepted *Custody* is forwarded onwards. The timer is associated with the *Link* on which the *Bundle* was forwarded. If the timer expires before this *BD* receives a *Custody Signal* from a new *Custodian* indicating that it has taken over *Custody*, this *BD* can decide to try and forward the *Bundle* again on the same *Link* if it is still *open*.	**4.6.5, 4.12.5**
D-Bus	Intra-machine inter-process message passing mechanism. Suggested for an alternative *API* implementation.	**4.4.2**
Deferred Bundles	A queue of *Bundles* waiting their turn to be transferred to the *Send Queue* of a *Link* so that they can be transmitted. Used by *Table-based Routers* as a way to limit the extra storage requirements needed by *Bundles* in the *Send Queue*; in the *Send Queue* both the internal and wire formats of the *Bundle* have to be stored, whereas in the *Deferred Bundles* queue only the master internal format is stored in memory.	**4.8.2**
Delivery Cache	Cache of information about *bundles* (recently) delivered a *registration*. The information is sufficient to identify duplicate bundles so that they can be suppressed from delivery.	**4.4.3**
Delivery Predictability	An estimate of the probability that a *Bundle* will be delivered to its destination if it remains in this *BD* node. Used in the **[PRoPHET]** routing algorithm.	**4.8.5**
Destination EID	Field in the *Bundle Primary Block* specifying the *EID* of a node to which the *Bundle* is to be delivered.	**4.4.1.1.12, [RFC5050]**
Diagnostic Interplanetary Network Gateway	Protocol for remote management of the *BD* over a transport using *Bundles* and a *MIB* to organize the data carried.	[Ding], **4.11.1**
DING	*Diagnostic Interplanetary Network Gateway*	[Ding], **4.11.1**

Term	Explanation	More details
	protocol.	
Discovery Agent	Component that manages *Announcements* advertising that a BD will accept communications using a specified *convergence layer* at a specified transport address.	**4.9.3.2**
DTN	Delay- and Disruption-Tolerant Networking.	**1**
DTN2	Reference implementation of a *BD* that implements a large part of the functionality described in this document, and more besides. Sponsored by *DTNRG.*	**1**
DTNRG	Internet Research Task Force *DTN* Research Group. Sponsors of the *DTN2* reference implementation from which this specification is derived.	**1**
Dynamic Routing	Category of *DTN* routing and forwarding mechanisms where routing decisions are influenced by information that is shared during opportunistic encounters of nodes (such as *Delivery Predictabilities*) in networks which have little or no predetermined topology rather than being controlled by a static table of routes or exchange of topology information in a network that has a relatively static topology.	**4.8.4**
EID	Endpoint Identifier.	**4.7**
EID Dictionary	Field in the *Bundle Primary Block* which carries strings representing the *EIDs* used in the various *Block* headers.	**4.6.2.3**
EID Pattern	A pattern used in *Registrations* and *Routes* against which *EIDs* can be matched to determine if a *Bundle* satisfies an identification constraint. For example, a *Registration* may be set to accept delivery of *Bundles* destined for a range of *EIDs*.	**4.4.3, 4.7.1, 4.8.2.2**
Endpoint	A *Bundle Endpoint*, usually known as just an *Endpoint* in this document and elsewhere, is a set of zero or more nodes supporting a *BD* to and from which *Bundles* can be sent. These nodes all identify themselves for all purposes associated with the *Bundle Protocol* by a single text string, the *Endpoint Identifier*. Note that a node may identify itself by multiple *EIDs* and hence be a member of multiple *Bundle Endpoints*.	**4.7**

Term	Explanation	More details
Endpoint ID	Endpoint Identifier	**4.7**
Endpoint Identifier	Identifier for a *Bundle Endpoint*. Takes the form of a Uniform Resource Identifier (URI) **[RFC3986]**. It is intended that all *EIDs* will be taken from the *dtn:* URI scheme currently under development in the *DTNRG*.	**4.7**
Epidemic all bundles list	See *All Bundles* list for *Flood Router*.	**4.8.3.2**
Epidemic Routing	See *Flood Routing*.	**4.8.3.2**
ESB	*Extension Security Block*.	[BSP]**, 4.6.1**
Ethernet Frame CL	Simple *Non-connection Oriented CL* using raw Ethernet frames to transport bundles in the local Ethernet network. Uses Ethernet MAC addresses as transport addresses.	**1.2**
Extension Block	Generic name for *Bundle Blocks* other than the *Primary Block* and the *Payload Block*. At present *DTN2* and this specification categorize *Extension Blocks* into *Metadata Blocks*, *Bundle Security Protocol Blocks*, and *'other'*. There are several sorts of *other* blocks defined already, and it may be necessary to improve this categorization in future.	**4.6**
Extension Security Block	Provides security for non-payload blocks in a bundle. ESB therefore is not applied to PIB or PCBs, and of course is not appropriate for either the *Payload Block* or *Primary Block*.	[BSP]**, 4.6.1**
External CL	*CL* for *DTN2* that operates in a separate process and communicates via *RPC* messages using an XML-encoding scheme.	**1.2**
External Router	*Router* for *DTN2* that operates in a separate process and communicates via *RPC* messages using an XML-encoding scheme.	**1.2**
File-based CL	*CL* that passes all *Bundle* information in platform operating system files. Not operational at Version 2.7.0 of *DTN2*.	**1.2**
Flood Routing	*Flood* or *Epidemic Routing* involves sending a copy of every *Bundle* that arrives at the *BD* to every open *Link* and forwarding a copy of every extant, non-expired *Bundle* on every *Opportunistic Link* that is discovered and opened.	**4.8.3.2**
FORWARD_ACTION	See *Action Type*.	**4.8.1**

Term	Explanation	More details
Forwarding	Process of dispatching a *Bundle* along a *Link* to the *Next-hop* node chosen by the *Routing* algorithm.	**4.8**
Forwarding Log	Record of the actions performed on a *Bundle* in the *BD*. This includes the reception or creation of the *Bundle*, delivery of the *Bundle* to a *Registration* and *Forwarding* of the *Bundle* on a specific *Link*. The log is maintained in time order, so that the last item in the log is always the most up to date with respect to a particular *Link*.	**4.6.6**
Fragment	Forwardable segment of a *Bundle*. See *Fragmentation*.	**4.6.4**
Fragmentation	*Bundles* with a *Payload Block* that is longer than one octet, can be fragmented into two *(Bundle) Fragments* that can be *Forwarded* separately. Each *Fragment* must have at least one octet in its *Payload Block*. The rules about which *Blocks* are placed in each *Fragment* are described in **[RFC5050]** and later in this document. On arrival at the *Destination Endpoint*, *Bundle Reassembly* is performed on the *Fragments* to reconstitute the original bundle.	**4.6.4**
Fragmentation Manager	Component of the *Bundle Factory* that handles splitting of *Bundles* into *Fragments* and reassembly of *Bundles* from received *Fragments*. Cope with both *Proactive Fragmentation* and *Reactive Fragmentation*.	**4.6.4**
Fragment Identifier	A 3-tuple (could be also considered as a 4-tuple) consisting of (Creation Timestamp = (Bundle Creation Time, Bundle Creation Sequence Number), Source EID, Destination EID) that is used to index the *Fragments* of a bundle so that all *Fragments* can be associated, e.g., when the *Bundle* is to be reassembled.	**4.6.4**
GBOFID	*Global Bundle Or Fragment Identifier.*	
Global Bundle Or Fragment Identifier	A 5-tuple (could be also considered as a 6-tuple) consisting of (Source EID ,	

Term	Explanation	More details
	Creation Timestamp = (Bundle Creation Time, Bundle Creation Sequence Number), Flag indicating *Fragment* or *Bundle*, Fragment length, Fragment offset) that is used to index *Registration Delivery Caches* and the *Table-based Router Reception Cache*.	
Idle Timer	Timer used to monitor *On Demand Links*. Restarted whenever data is sent or received on the *Link*. If the timer expires, the *Link* is closed but will reopened if needed for more data to be sent (from either end).	**4.9.1.4, 4.9.4**
IETF	Internet Engineering Task Force	
Interface	A *Bundle* transport communications end point associated with a *Convergence Layer*. For *Connection Oriented CLs*, the *Interface* is used to provide a passive listener through which connections can be established. For *Non-connection Oriented CLs* the *Interface* can be used to receive *Bundles*.	**4.9.1.3, 4.9.2**
Interface Manager	Component that maintains a table of current *Interfaces*.	**4.9.2**
Internet Draft	Pre-standard draft document published by the Internet Engineering Task Force.	
IP	Internet Protocol.	
IRTF	Internet Research Task Force	
Keepalive (Message)	Short message sent over *Connection Oriented CLs* to confirm liveness if no data or other messages have been sent for a significant period of time. If no messages are received for several *Keepalive Messages* periods, the *Link* is assumed to be broken and is closed.	**4.9.6.2**
Key Database	Database of security *Keys* maintained for use by the *BSP*.	**4.6.3**
LEID	See *Local EID*	**4.7**
LTP	Licklider Transmission Protocol.	**1.2**
Licklider Transmission Protocol	Transport protocol designed to provide retransmission-based reliability over links characterized by extremely long message round-	**1.2**

Term	Explanation	More details
	trip times (RTTs) and/or frequent interruptions in connectivity.	
Link	Potential (available or unavailable) or Actual (opening or open) communication channel using a specific *Convergence Layer* to a *Next-hop* node.	**4.9.1.4, 4.9.4**
Link set	Currently extant *Links* known by the *Contact Manager*.	**4.9.4, 4.9.8.1**
Local EID	*EID* allocated to the current *BD*. *DTN2* at present only allows one *Local EID* per *BD*.	**4.7**
LTP CL	*Convergence Layer* based on *LTP*. Under development and expected to be incorporated in *DTN2* during 2010.	**1.2**
Management Information Base	Scheme for structuring management and configuration information used for the *DING* protocol.	**4.11.2, 4.11.6**
Management Component	Implements the set of management and configuration commands that can be used to control the *BD*.	**4.11**
Maximum Transmission Unit	Constraint on the size of data unit that can be transmitted over a *Convergence Layer*.	**4.5.5, 4.6.4, 4.11.7.6**
Metadata	Used to describe the set of *Delivery Predictabilities*, *Bundle* offers and acceptances exchanged between a pair of nodes encountering opportunistically when using the PRoPHET and many other *Dynamic Routing* strategies. They are *Metadata* because they describe the data to be exchanged and the criteria for exchanging the actual data.	**4.8.5**
Metadata Block	*Bundle Protocol Extension Block* is designed to carry additional information that DTN nodes can use to make processing decisions regarding bundles, such as deciding whether to store a bundle or determining to which nodes to forward a bundle.	**4.6**
MIB	Management Information Base	**4.11.6**
MTU	*Maximum Transmission Unit*.	**4.5.5, 4.6.4, 4.11.7.6**
Multinode Endpoint	Situation where several nodes use the same *DTN Endpoint Identifier*. See *Singleton*	**4.4.1.1.12, 4.7**

Term	Explanation	More details
	Endpoint.	
N4C	Networking for Communication Challenged Communities. The EU Framework Programme 7 project that supported the creation of this document.	
Next-Hop	The peer node at the other end of a *Link*. The *BD* that will receive a *Bundle* forwarded on that *Link*. This *BD* will be the *Previous Hop* when the *Bundle* is received.	**4.9.1.4**
Next-Hop Route	*Route Table Entry* that explicitly specifies a *Link* to use to reach the *Next-Hop* node. See also *Waypoint Route*.	**4.8.2**
Non-connection Oriented CL	*CL* in which the underlying transport protocol does not offers a connection-based service. The transport is unlikely to offer a reliable service and can make no guarantees about in order delivery.	**4.9.1.2.1,**
NORM CL	NACK-Oriented Reliable Multicast over IP *Convergence Layer*.	**1.2**
Null EID	Place holder *EID*. For the dtn: URI scheme this is *dtn:none*.	**4.7**
Object Identifier	Hierarchical Numbering Scheme used in Network Management *MIBs*. Used to identify information carried in *DING*.	**4.11.2**
OID	*Object Identifier*.	**4.11.2**
On Demand	Type of *link* that has a *contact* that is opened when there is data to transmit, if the underlying transport can provide it, and closed again a period of time after the last data transmission (see *Idle Timer*).	**4.9.1.4**
Opportunistic	Type of *link* that opens a *contact* either in response to a direct request from the proposed peer or as a result of the reception of an *Announcement* from the proposed peer. Once opened the *Contact* remains open until explicitly closed or communication with the peer is broken.	**4.9.1.4**

Term	Explanation	More details
Passive Registration	*Registration* to which *bundles* cannot currently be delivered.	**4.4**
Payload	The user data or PDU carried in a *Bundle*.	**4.6**
Payload Block	The *Block* that encapsulates the *Payload* in a *Bundle*. A *Bundle* has either zero or one *Payload Blocks* and the *Payload* has to have at least one octet of data.	**4.6**
Payload Confidentiality Block	indicates that the payload has been encrypted, in whole or in part, at the PCB *security-source* in order to protect the bundle content while in transit to the PCB *security-destination*.	[BSP], **4.6.1**
Payload File	Platform operating system file used as persistent storage for the contents of the *Payload Block* of a *Bundle*.	**4.6**
Payload Integrity Block	Used to assure the authenticity and integrity of the payload from the PIB *security-source*, which creates the PIB, to the PIB *security-destination*, which verifies the PIB authenticator.	[BSP], **4.6.1**
Payload Security Block	Old name for *Payload Integrity Block*.	[BSP], **4.6.1**
PCB	*Payload Confidentiality Block*.	[BSP], **4.6.1**
PDU	*Protocol Data Unit*.	
Peer	The partner node at the other end of a *Link*. Also known as *Next-Hop* or *Previous Hop* depending on context.	**4.9.1.4**
Pending Bundles Queue	Queue of *Bundles* maintained by the *Bundle Core* that are eligible for *forwarding* if the *Router* determines it is appropriate.	**4.5.1**
Persistent Storage	Permanent storage used to hold *Bundles*, etc. to allow the *BD* to be shutdown and restarted without loss of information.	**4.10**
Potential Downtime	Conservative estimate of the maximum amount of time that a link is expected to be 'down' after an unexpected closure during normal operation. May be used by some types of routing to decide if a *Bundle* should be rerouted if the contact is not re-established within this time.	**4.9.4.1,**

Term	Explanation	More details
Previous Hop	The peer node at the other end of a *Link*. The *BD* that sent a *Bundle* received on that *Link*. This *BD* will be the *Next-Hop* for the *Bundle* as seen from the sending node.	**4.9.1.4**
Previous Hop Block	*Bundle Protocol Extension Block* used to carry the *EID* of the node that last forwarded the *Bundle* to this *BD*. Useful with *Non-connection Oriented CLs* that may not be able to supply the *Previous Hop EID*. May also be used when a forwarding node has multiple *EIDs* to indicate which *EID* is deemed to be the *Previous Hop*. This *Block* must only be carried for one *DTN* hop; the recipient must delete the *Block* from the *Bundle*, but may add a new one when forwarding the *Bundle* again.	**4.6**
Primary Block	Master *Block* that must be the first *Block* in a *Bundle*.	**4.6**
Proactive Fragmentation	*Fragmentation* of a *Bundle* in advance of transmission to satisfy known constraints on the size of a bundle due to the *MTU* of the *CL* or anticipated duration of a *Contact*.	**4.6.4**
PRoPHET Routing	*DTN Dynamic Routing* mechanism using *Delivery Predictabilities*.	**4.8.5**
Protocol Data Unit	The data of the next higher layer protocol carried in a protocol message.	
PSB	*Payload Security Block*.	[BSP], **4.6.1**
Reactive Fragmentation	*Fragmentation* of a *Bundle* as a result of a transmission failure resulting in a partial transmission of a *Bundle* by a *Convergence Layer*. Can only be carried out if at least one octet of the *Payload Block* data has been transmitted, and should only be done if the sender and receiver have reliable information about what has been received out of the data transmitted.	**4.6.4**
Reception Cache	Information about *Bundles* recently received by a *Table-based Router* sufficient to allow identification and suppression of duplicates.	**4.8.2**

Term	Explanation	More details
Registration	Expression of interest in *Bundles* with a *Destination EID* that matches a *Pattern EID*. When an application is running, it may make a *Registration* into an *Active Registration* that results in *Bundles* with *Destination EIDs* that match the *Pattern EID* being delivered to the application. Otherwise *Registrations* are *Passive Registrations*.	**4.4, 4.4.3**
Registration Expiration Timer	Timer created to manage the lifetime of a *Registration*. When it expires, the *Registration* will be deleted the next time that it becomes a *Passive Registration*.	**4.4.3**
Registration ID	Unique number identifying a *Registration*. Preserved across *BD* shutdown and restart until the *Persistent Storage* database is reinitialized.	**4.4**
Registration Table	Table of all extant *Registrations*.	**4.4.3**
Remote EID	*EID* of the peer node at the remote end of a *Link*.	**4.9.4**
Replyto EID	*EID* (including *Service Tag)* of node to which *Status Reports Bundles* should be sent.	**4.4.1.1.12, [RFC5050]**
Reroute Timer	Timer associated with an *Always On* or *On Demand* link where communication failure has been detected. In case communication is restored in a short time (see *Potential Downtime)* rerouting of *Bundles* queued for transmission on the *Link* is postponed until the timer expires, thereby helping to avoid the route flap phenomenon.	**4.8.2**
RFC	Request for Comments. IETF and IRTF publications containing standards and related documents.	
RFCOMM CL	*Bluetooth Connection Oriented Convergence Layer*.	**[RFCOMM], 1.2**
Route Entry	A *Next-Hop Route* or *Waypoint Route* in the *Routing Table*.	**4.8.2**
Router	Component that decides on which *Bundles* in the *Pending Queue* should be forwarded on which *Links* according to a defined routing algorithm and mechanism. This may depend on *Route Entries* configured by *BD* configuration commands as in the basic *Table-based Router* or	**4.8**

Term	Explanation	More details
	a routing data exchange protocol such as *DTLSR*; or it may depend on dynamically acquired information as in the *PRoPHET Router*.	
Routing Table	Table of *Route Entries* used by *Table-based Routers* when making forwarding decisions.	**4.8.2**
RPC	Remote Procedure Call. An inter-process communication protocol originally developed by Sun Microsystems.	**4.4.1**
Scheduled	Type of *Link* that opens a *Contact* according to a predefined schedule.	**4.9.1.4**
Scheduled Routing	Form of routing that relies on an 'oracle' to inform it when *Bundles* intended for a particular *Destination* EID should be sent to a specified *Next-Hop*. Essentially unimplemented in DTN2.	**1.2**
Scheme Part	The part of a URI before the first colon (':') separator. See *Uniform Resource Identifier*.	**4.7**
Scheme Specific Part	The part of a URI after the first colon (':') separator. See *Uniform Resource Identifier*.	**4.7**
SDNV	*Self-Delimiting Numeric Value*.	**[RFC5050]**
Security Block	One of *BAB*, *PIB*, *PCB* or *ESB*, (also *CB* or *PSB*).	
Security-Destination	A *Bundle Node* that processes a security block of a *Bundle*.	[BSP], **4.6.1**
Security Policy Database	Repository of information indicating what *BSP* mechanisms should be applied when sending *Bundles* to a particular *Destination EID* and what security mechanisms should have been applied on incoming *Bundles*.	**4.6.3,**
Security Policy Enforcement Point	The *BD* provides enforcement of security as specified by the *SPD* and the *BSP* for *Bundles* received and transmitted by the *BD*, i.e., it provides a *Security Enforcement Point*.	**4.6.3**
Security-Source	A bundle node that adds a *Security Block* to a *Bundle*.	[BSP], **4.6.1**

Term	Explanation	More details
Self-Delimiting Numeric Value	Mechanism defined in **[RFC5050]** for expressing arbitrarily large positive integers and bit patterns in protocol fields in a format that does not require advance specification of the maximum value that can be stored.	**[RFC5050]**
Serial Link CL	*Connection Oriented Convergence Layer* running over point to point communication paths using RS-232 or similar serial line protocols.	**1.2**
Service Tag	String concatenated with an *EID* for a node (probably part of the URI *path* component) to provide a demultiplexing mechanism for *Bundles* arriving at a node that is identified by the *EID*. The demultiplexing associates *Bundles* with *Destination EIDs* that *subsume* a *Local EID* with a *Registration* that expresses a *Pattern EID* which matches the *Destination EID*. The simplest form of a Pattern EID is a concatenation of a *Local EID* (typically taken from the *dtn:* scheme) with a forward slash ('/') and a fixed string *Service Tag* without any wildcard characters ('*') embedded in it. More complex *Pattern EIDs* may contain one or more wildcard characters so that they may match a selection of *Destination EIDs*.	**4.4, 4.4.3**
Simple Network Management Protocol	Protocol used in the conventional Internet to carry Network Management information.	**4.11.1**
Singleton Endpoint	Situation where a *DTN Endpoint Identifier* is used for exactly one node. See *Multinode Endpoint*.	**4.4.1.1.12, 4.7**
SNC	Sámi Networking Connectivity. Predecessor project to N4C.	
SNMP	*Simple Network Management Protocol*.	
Source EID	Field in the *Bundle Primary Block* specifying an *EID* of the node from which the *Bundle* has been sent. May be the *Null EID*..	**4.4.1.1.12, [RFC5050]**
SPD	*Security Policy Database*.	**4.6.3**
SPEP	*Security Policy Enforcement Point*	**4.6.3**
Static Routing	Uses a *Table-based Router* with the *Route Table* filled only by configuration commands. Therefore the routing provided is all statically defined.	**4.8.3.1**
Status Report	*Administrative Bundle* with the data format of a	**4.5.5**

Term	Explanation	More details
	Status Report as defined by **[RFC5050]** as its *Payload*.	
Store, Carry and Forward	Fundamental paradigm of *DTN*.	**1, 4.4**
Subsumed EID	An *EID* which contains a *Local EID* as an initial substring, i.e., it is a *Local EID* with a (generally) non-empty *Service Tag* attached.	**4.7**
Suppress	Used in the context of routing and forwarding when the forwarding of some *Bundles* with specific characteristics is not allowed, e.g., because the information they contain is stale.	**4.6.6, 4.8.1**
Table-based Router	Router model in which forwarding of a *Bundle* is predominantly controlled by searching for a suitable *Next-Hop* in a previously defined table of *Route Entries*. A number of different routing mechanisms use the basic *Table-based Router* as a starting point, but fill the table is different ways.	**4.8.2**
TCP	Transport Control Protocol - Connection oriented transport protocol in the IP suite.	
UDP	User Datagram Protocol. Part of the IP suite.	
Uniform Resource Identifier	Structured identifier as used for naming Endpoints. Constructed from a *Scheme* name (DTN expects to concentrate on the DTN-specific *Scheme dtn:* being defined in **[URIscheme]** and **[URIfind]**) and a *Scheme Specific Part* being the remainder of the URI.	**4.7**
URI	*Uniform Resource Identifier*.	**4.7**
URI Metadata Type	*Payload* for a *Metadata Extension Block* containing one or more *URIs* as null terminated strings.	**4.6**
Waypoint	Node specified by (one of) its *EIDs* that represents a suitable intermediate destination for a *Bundle* being sent towards a given *Destination EID*.	**4.8.2**
Waypoint Route	*Route Entry* defining a useful intermediate destination for a *Bundle* on its way to a given *Destination EID*. The *Table-based Router* performs an iterative search on the *Route Table* in order to find a *Next-Hop Route* that leads through a sequence of *Waypoints* to the *Destination EID*.	**4.8.2**

Term	Explanation	More details
Wi-Fi/802.11	Wireless local area networking technology.	
Wildcard EID	An *EID Pattern* that matches any conceivable *EID* from any scheme - it is the string **:**.	**4.8.3.2**
XDR	EXternal Data Representation. Marshalling and unmarshalling scheme used by the Sun *RPC* inter-process communication mechanism.	**4.4.1**
XML	Extensible Markup Language.	**1.2**

2.5 REFERENCES

[BSP]	Symington, S., Farrell, S., Weiss, H., and Lovell, P., *"Bundle Security Protocol Specification"*, draft-irtf-dtnrg-bundle-security-15, February 2010.
[Bytewalla]	Bytewalla Project Team, *"Bytewalla: Delay Tolerant Network on Android Mobile Phones"*, Bytewalla project web site, January 2010
[CBHE]	Burleigh, S., *"Compressed Bundle Header Encoding (CBHE)"*, draft-irtf-dtnrg-cbhe-04, February 2010.
[Checksum]	Eddy, W., Wood, L., and Ivancic, W., *"Reliability-only Ciphersuites for the Bundle Protocol"*, draft-irtf-dtnrg-bundle-checksum-06, October 2009.
[Dbus]	Pennington, H., Carlsson, A., Larsson, A., *et al*, *"D-Bus Message Bus System"*, Web Site and Wiki.
[Ding]	Clark, G., Campbell, G., Kruse, H. and Ostermann, S., *"DING Protocol -- A Protocol For Network Management"*, draft-irtf-dtnrg-ding-network-management-02, February 2010.
[DTLSR]	Demmer, M. and Fall, K. *"DTLSR: Delay Tolerant Routing for Developing Regions"*, ACM SIGCOMM Workshop on Networked Systems for Developing Regions (NSDR), August 2007, Paper available.
[DTN2]	Demmer, M., *et al*, *"DTN2 Bundle Protocol Agent Reference Implementation"*, Available from Sourceforge DTN Repository or N4C Code Repository.
[DTNmib]	Bers, J., *"DTNmib Module"*, BBN Technlogies/NASA, February 2010
[DTNRG]	Internet Research Task Force (IRTF) Delay- and Disruption-Tolerant Research Group Web Site and Wiki.
[DTNstateArt]	Davies, E, *et al*, *"DTN - The State f the Art"*, N4C Deliverable D2.1, April 2009,
[DTNnetMgmt]	Ivancic, W., *"Delay/Disruption Tolerant Networking - Network Management Requirements"*, draft-ivancic-dtnrg-network-management-reqs-00, June 2009.
[ECOS]	Burleigh, S., *"Bundle Protocol Extended Class Of Service (ECOS)"*, draft-irtf-dtnrg-ecos-00, December 2009.
[Inquiry]	Bluetooth Special Interest Group, *"Bluetooth Communication Topology - Inquiry Procedure"*, Bluetooth SIG Web Site.
[MetadataBlock]	Symington, S., *"Delay-Tolerant Networking Metadata Extension Block"*, draft-irtf-dtnrg-bundle-metadata-block-07, February 2010.
[N4Carch]	Davies, E, *et al*, *"N4C System Architecture"*, N4C Deliverable D2.1, April

	2009,
[PrevHopBlock]	Symington, S., *"Delay-Tolerant Networking Previous Hop Insertion Block"*, draft-irtf-dtnrg-bundle-previous-hop-block-11, February 2010.
[PRoPHET]	Lindgren, A., Doria, A., Davies, E., and Graši , S., *"Probabilistic Routing Protocol for Intermittently Connected Networks"*, draft-irtf-dtnrg-prophet-05, February 2010.
[PubSub]	Demmer, M., and Fall, K., *"The Design and Implementation of a Session Layer for Delay Tolerant Networks "*, Elsevier Computer Communications, Feb 2009
[RetransBlock]	Symington, S., *"Delay-Tolerant Networking Retransmission Block"*, draft-irtf-dtnrg-bundle-retrans-block-06, October 2009.
[RFC3986]	Berners-Lee, T., Fielding, R., and Masinter, L., *"Uniform Resource Identifier (URI): Generic Syntax"*, STD 66, RFC 3986, January 2005.
[RFC5050]	Scott, K. and Burleigh., S., *"Bundle Protocol Specification"*, RFC 5050, November 2007.
[RFC5325]	Burleigh, S., Ramadas, M., and Farrell, S. , *"Licklider Transmission Protocol - Motivation"*, RFC 5325, September 2008.
[RFC5326]	Ramadas, M., Burleigh, S., and Farrell, S., *"Licklider Transmission Protocol - Specification"*, RFC 5326, September 2008.
[RFC5327]	Farrell, S., Ramadas, M., and Burleigh, S., *"Licklider Transmission Protocol - Security Extensions"*, RFC 5327, September 2008.
[RFC5531]	Thurlow, R., *"RPC: Remote Procedure Call Protocol Specification Version 2"*, RFC 5531, May 2009.
[RFCOMM]	Bluetooth Special Interest Group, *"RFCOMM - How it works"*, Bluetooth SIG Web Site.
[SNCPRoPHET]	Graši , S., *et al.*, *"Implementation of PRoPHET DTN Routing Protocol"*, Available from N4C Code Repository, as made for Sámi Network Connectivity project.
[TCPclayer]	Demmer, M. and Ott, J., *"Delay Tolerant Networking TCP Convergence Layer Protocol"*, draft-irtf-dtnrg-tcp-clayer-02, November 2008.
[TCA]	Set, A., Darragh, P., Liang, S., Lin, Y. and Keshav, S., *"An Architecture for Tetherless Communication"*, University of Waterloo, Paper, July 2005.
[TCAhowTo]	Tetherless Computing Laboratory, *"Installing and Running TCA Router"*, University of Waterloo, Wiki, 2005.

[UDPclayer]	Kruse, H. and Ostermann, S., *"UDP Convergence Layers for the DTN Bundle and LTP Protocols"*, draft-irtf-dtnrg-udp-clayer-00, November 2008.
[URIfind]	Davies, E. and Doria, A., *"Adding the "find" Operation to the dtn: URI Scheme"*, draft-davies-dtnrg-uri-find-01, October 2009.
[URIscheme]	Fall, K., Burleigh, S., Doria, A., and Ott, J., *"The DTN URI Scheme"*, draft-irtf-dtnrg-dtn-uri-scheme-00, March 2009.

3. EXTERNAL INTERFACES OF BUNDLE DAEMON

The Bundle Daemon (BD) does not have any direct interfaces to human users. It is intended as a piece of middleware which provides the interface between applications which make use of bundles for data transfer and the operating system of the hosting node. The daemon is implemented so that it can be a permanently running process to which applications can connect to send and receive bundles. However, the daemon is also capable of terminating and restarting without loss of state. This feature caters for the expected behaviour of many of the systems on which the BD will be deployed where power consumption is a vital consideration. Important internal state is kept permanently backed up to persistent storage so that the daemon can cope with both graceful and unexpected termination without significant loss of state data, allowing it to be restarted when appropriate and be able to recover all the state information during restart. An application programming interface (API) is provided for connecting applications to the daemon process. External management of the system is carried out over the same API. The BD requires networking and file system functionality from the host operating system (OS). The external interfaces are illustrated in Figure 2.

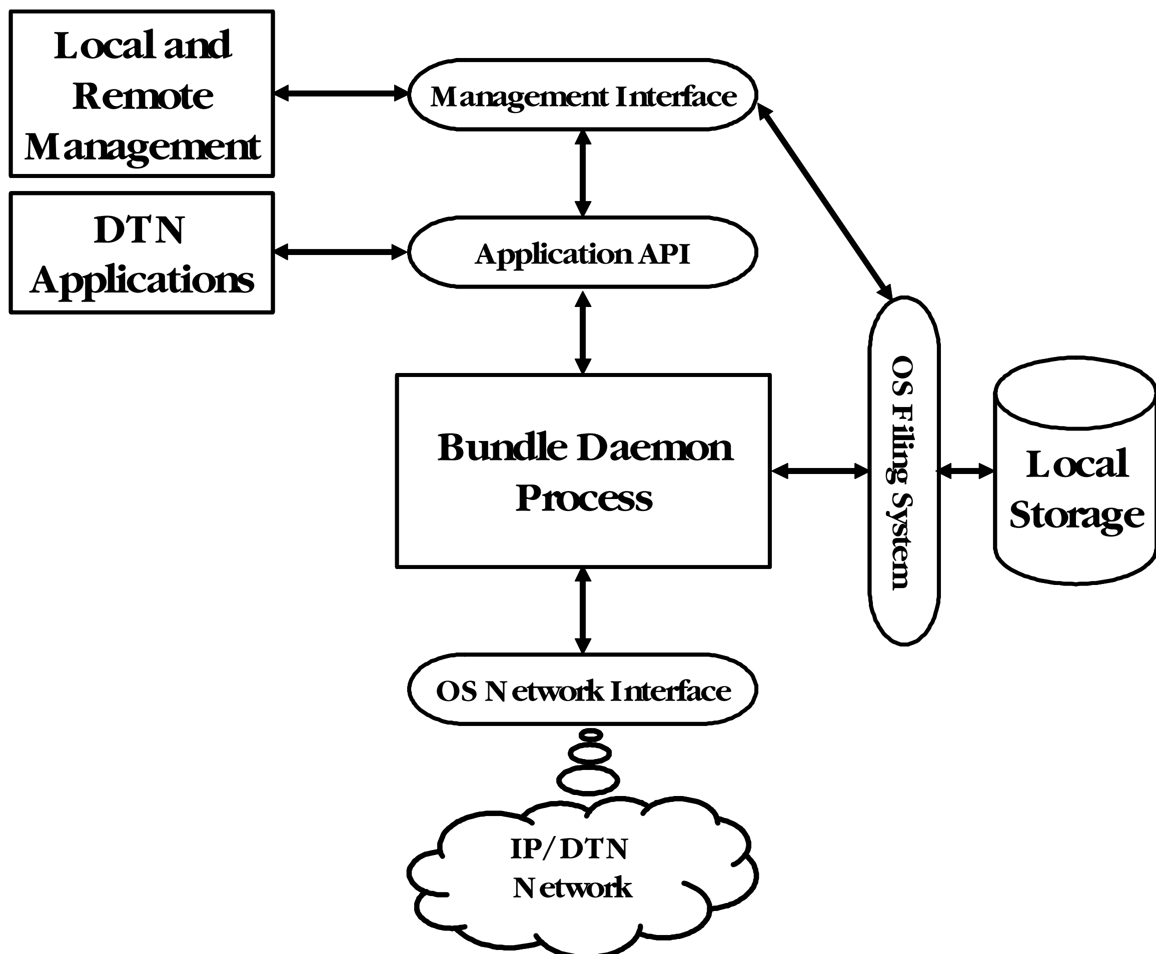

FIGURE 2 EXTERNAL INTERFACES OF BUNDLE DAEMON

The details of the various interfaces are described in Section 4.

The primary Application API is a Sun RPC [RFC5531] based interface that is compatible with the interface provided by the DTN2 reference implementation. The initial version of the BD as specified here may not support some, as yet, un-standardized features of DTN2 but will provide dummy parameters in the API for compatibility purposes where these have been implemented in DTN2 to provide access to the extra features (particularly the Publish/Subscribe mechanism).

The primary API may not be ideal for some purposes. Some of the perceived issues are described here. It is essentially a half-duplex modal interface, in that the interface has to specifically be switched into a special mode in order to receive bundles and there is no asynchronous notification mechanism available to inform an application that a bundle has been received. Additionally the interface was designed when the Bundle Protocol had a limited number of block types that were of interest to the application. With the advent of several new auxiliary block types, the existing interface is somewhat clumsy as a means of adding such extra blocks to a bundle. It is proposed to add a secondary application API based on the D-Bus technology that will resolve some of these problems and provide a more modern interface style for the BD. Note that using a D-Bus interface restricts the use of the BD to applications that are also hosted on the machine whereas the RPC interface can be used from other hosts over a local network. In practice and especially for N4C applications, this capability has not been significantly exploited, though by providing both the SUN RPC and the D-Bus APIs, the capabilities will be available if needed. The secondary API is under development and will be documented in a later version of this functional specification.

Additional API capabilities may be needed in future to configure and use the security policy databases and the key database associated with the Bundle Security Protocol [BSP]. At present there are limited capabilities in the management interface for this purpose (see Section 4.11.7.13).

The Network Interface has a number of sub-components that access the (IP) network for the following purposes:

- IP-based neighbour node discovery: Compatible with the DTN2 reference implementation IP discovery mechanism using UDP to a well-known multicast group address to identify nodes that have come into communication range and may be interested in establishing an opportunistic link with a view to exchanging DTN bundles.
- TCP over IP bundle exchange link: Link establishment and management mechanism compatible with DTN2 allowing for the establishment of a unicast TCP connection between consenting partners in an opportunistic encounter for the exchange of bundles.
- UDP over IP bundle exchange link. This is a very rudimentary system allowing smallish bundles (notionally up to 64 kilobytes, but in practice more usually 8 kilobytes because of IP fragmentation issues) to be exchanged.
- TCP over IP dynamic routing protocol metadata exchange link: Dynamic routing protocols such as PRoPHET need to exchange ‘metadata’ regarding delivery probabilities, prioritised lists of candidate bundles that are ready for transfer and other protocol specific data. It might be possible for this metadata to be encapsulated in bundles and exchanged over the bundle exchange link between

nodes involved in an opportunistic encounter. However the need for closer control over the link and the need for periodic retransmission makes the use of bundles somewhat clumsy and may necessitate 'layer violations' whereby the dynamic routing protocol has to control the bundle convergence layer directly.

The N4C BD implementation may in future be extended to utilise Bluetooth or other network interfaces and is written to make it easy to add additional network interfaces.

The Filing System Interface will be used to provide persistent storage of bundles and other information needed to support seamless shutdown and restart. Management of the persistent storage will be carried out using a 'standard' database package. In the current DTN2 implementation the Berkeley DB system is the default choice but an SQL based system such as MySQL or Postgres can be used as an alternative. The simple SQLite database is intended as the default choice for N4C implementations. [DTN2 allows user selection of the database interface to be used. In future, particularly for low end machines, the SQLite implementation of an SQL database will probably be most useful.]

The BD must be capable of being shutdown and restarted in a way that allows operations to resume, as far as possible, as if the BD had been running continuously. The persistent storage of information is vital to this process, and the persistent store has to be kept updated as far as possible at all times so that unexpected shutdowns result in minimal data loss.

4. FUNCTIONALITY

The functionality to be provided in the Bundle Daemon (BD) can be divided into a number of components. The BD is an event driven system that reacts to events from a number of sources:

- External events stemming from actions on the external interfaces.
- Internal events resulting from notifications sent by other components.
- Timer events such as bundle expiry and idle timeouts.

Each component will 'consume' a number of events and, in most cases, generate consequent events that will be routed to other components as appropriate. These components will generally function as state machines driven by the incoming events; some of the state may, for example, be on a per bundle or per connection basis rather than a single global state for the component, so that events may be directed to particular items managed by a component.

For some components, a BD may contain multiple instantiations of a component that are based on a common object framework but implement the functionality for different schemes (e.g., different dynamic routing protocols, different transport convergence layers). It is envisaged that these components may be dynamically instantiated according to the configuration requested by the node administrator and relevant connections. For such components there will be an additional manager component that handles the dynamic instantiations.

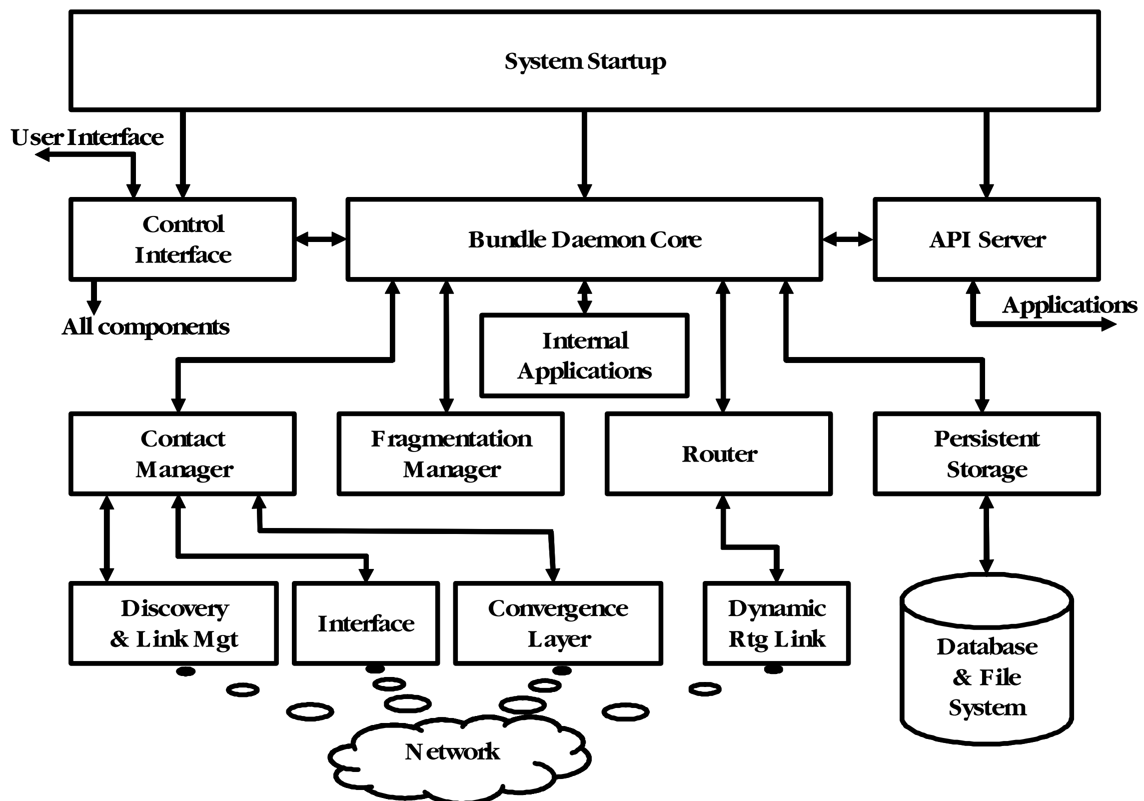

FIGURE 3 BUNDLE DAEMON ARCHITECTURE

The system architecture shown in Figure 3 comprises the following components:

- System Start-up Controller
- DTN Application Interface Server
    - Primary API (compatible with DTN2 at release 2.6.0)
    - Secondary API (based on D-Bus)
- Bundle Daemon Core
    - Event Distribution
- Bundle Factory and Fragmentation Manager
    - Bundle Construction, Dissection and Validation
    - Bundle Security Protocol Support
    - Bundle Fragmentation
    - Bundle Custody and Custody Timers
    - Forwarding Information and Forwarding Log
- Endpoint Identifier Management
    - Parsing Endpoint Identifiers
- Bundle Router
    - Table-based Routing Manager
        - Static Routing
        - Epidemic Routing
    - Dynamic Routing Manager
        - PRoPHET Routing
- Contact Manager
    - Interface Manager
    - Discovery Manager
        - IP Discovery
        - Bonjout Discovery
    - Link Management
    - Convergence Layer Manager
        - TCP Convergence Layer
        - UDP Convergence Layer
        - Null Convergence Layer
- Storage Manager
    - Global State (overall configuration items)
    - Bundle State
    - Registration State
    - Link State
    - PRoPHET Dynamic Routing State
- Configuration and Management
    - Management API
    - Configuration Change Notification
    - Logging and Log file Management
    - Management Information (MIB)
    - Management and Configuration Commands

- Internal Applications
  - DTN Ping Echo Application
  - DTN Traceroute Application
  - DTN Management Interface Application
  - DTN Heartbeat Application
  - Administrative Bundle Handler
  - External Router Application

These components will be described in the following sections including descriptions of

- The state to be maintained in each
- The events that affect the component and the actions to be done when the event is received
- The event notifications generated as a result of the actions
- Any external interfaces managed by the component.

Individual units of functionality have been numbered and there is extensive cross referencing of the items. This will allow implementation and testing to be traced back to this specification.

4.1 LOGGING

The whole system should provide extensive, consistent, configurable logging for all components.

1. **Logging configuration.** Logging will be controlled by a logging specific configuration file.

2. **Identifying the Logging Configuration file**. The Logging control configuration file may be changed from the hard coded default by specifying an environment variable or command line argument.

3. **Logging should provide several levels of logging**. Levels will be at least ALWAYS, CRIT, ERR, WARN, INFO, DEBUG. Every logging message will be associated with an appropriate level. Logging levels will be nested so that enabling logging at a given level will result in all messages at lower levels being output (where ALWAYS is the lowest level and DEBUG the highest). The implication is that logging will be more verbose at higher levels whereas lower levels will confine themselves to imparting vital operational information.

4. **Fine control of logging**. For each component (or, at implementers' discretion, part of a component), the logging system should allow the logging level to be controlled individually. This control will be hierarchical with the specifiers that control the logging defined hierarchically so that a logging level control affects all logging at lower levels of the hierarchy unless it is specifically altered at the lower level. [DTN2 uses a filesystem-like hierarchical naming system to achieve this functionality.]

5. **Destination of log**. The log will be written to a file specified in the logging configuration file.

6. **Log rotation**. The logging file may be emptied and logging restarted either by a user control command or by signalling the process to allow rotation of log files.

7. **Log format.** Every log entry will contain the date and time of generation in a human readable format [DTN2 currently uses a format which is not human friendly.], the logging specifier in the logging statement that generated the message and the logging level that resulted in the message.

4.2 TEXTUAL REPRESENTATIONS OF INTERNAL STRUCTURES

For the purposes of management and monitoring, components provide a 'dump' function to create a textual representation of internal state that can be presented to a user/manager. These functions are not explicitly defined in the individual components, but it can be seen which components need to provide this functionality in the list of management functions in Section 4.11.7.

4.3 SYSTEM STARTUP CONTROLLER

The BD will be implemented as a multithreaded application with primary permanently running threads for:

- The Main Bundle Daemon Core component (referred to as the *bundle core*) handling all communication with the network and timers;
- The Application Programming Interface Server handling RPC-based (and in the future D-Bus-based) communication with applications using the BD; and
- The User Control Interface for the BD providing configuration and monitoring of the operations of the BD.

The System Start-up Controller manages the start-up of all components and the creation of all relevant threads. This includes reading the initial configuration, distributing it to all the interested components and invoking the restoration of the internal state of the BD from persistent data stored by a previous invocation.

The primary threads will be initially blocked to ensure that initialisation can be carried out in a controlled order. Once all initialisation is completed successfully, the threads are unblocked and started. Thereafter functions are driven by events generated as a result of input on the various interfaces: API commands, link advertisements, link connections, bundle transfers, dynamic routing metadata transfers and user control interface commands together with internal timer and management driven events.

Shutdown of the system is triggered either by a User Interface command or an external Operating System signal. Shutdown is managed within the Bundle Daemon Core which ensures that the persistent data images of the internal BD state are up to date, shuts down the interfaces in a controlled fashion and terminates the threads.

4.3.1 Detailed Functionality

8. **Start the logging system**.

9. **Read command line arguments**.

10. Initialise thread system with all threads blocked.

11. **Create Bundle Daemon Core**. Initialize all components, including mechanisms for handling configuration commands (see Section 4.11.7.3).

12. **Create API Server thread if required** (see Section 4.11.7.1).

13. **Read and action the initial configuration file supplied by user**.

14. **Initialize the persistent storage interface.** If requested by command line arguments either empty the datastore and create it anew, or tidy it up to remove existing data. Otherwise just prepare to read in data from previous sessions if any.

15. **Release the block on threads**.

16. **Run until Bundle Daemon Core terminates**. See item 486 for *Shutdown* command. Then close down all components and exit.

4.4 DTN APPLICATION INTERFACE SERVER

The DTN application interface is somewhat unusual as an application network interface because the application interface in the BD has persistent state related to the 'addresses' to which bundles are delivered. This is in line with the *Store, Carry and Forward* paradigm of DTN, in that the BD should be able to store bundles for an application until it is ready to take delivery, even if the application is not running when the bundle arrives - it would be counterintuitive (and potentially very bad for the performance of a DTN) if all bundles that were delivered to a node were immediately deleted unless the target application was actually running at the time the bundle was delivered, in contrast to the behaviour at intermediate forwarding nodes which wait for an opportunity to forward to the destination node until the bundle's expiry time passes.

The DTN API component provides this capability by managing a set of persistent *registrations* each of which is allocated a unique *registration id* Most registrations are created and/or deleted by an application that wishes to have bundles with a destination EID that combines a local EID allocated to this node with an appended *service tag* (demultiplexing tag) delivered to it. Registrations are created with a lifetime that may potentially be longer than the lifetime of the application creating the registration. If the application does not explicitly delete the registration when it terminates, the registration will persist until its lifetime runs out. Exactly one registration can be associated with an EID (including the service tag) at a given time.

Registrations are either *passive* or *active*. A registration can only be active if an application is *bound* to the registration so that bundles that match the EID encoded in the registration are delivered to the application. A registration will otherwise be passive; a registration bound to an application can be switched between active and passive under the control of the application. The registration can be configured so that if a bundle arrives that matches the registration EID arrives when the registration is passive, one of three things happens:

- The bundle may be immediately dropped if the BD is configured to allow 'early deletion' of bundles (default). Otherwise the bundle will just be kept until it expires at the end of its lifetime.
- The bundle is stored for deferred delivery when some application activates the registration or until the bundle expires at the end of its lifetime.

- The BD runs a specified executable script that will be assumed to start an application that will activate the registration. [The security implications of script execution need to be investigated]. Whether or not the executable script activates the registration immediately, the bundle will be retained as if deferred delivery had been requested.

The API component manages the persistent storage of registration state using the Storage Manager (Section 4.10.3).

The API component is notified by the BD core (Section 4.5.4, Item 56) whenever a bundle is received that matches any of the EIDs associated with this node. If a registration exists that matches the destination EID of the incoming bundle, the bundle will be examined and its disposition determined (deliver immediately if there is an active registration with a matching EID and an outstanding receive request, store for deferred delivery if there is either an active registration or a passive registration marked for deferred delivery, or otherwise dropped either immediately, if early deletion is allowed, or on normal expiry of its lifetime). The Storage Manager Component (Section 4.10) is informed accordingly and, in the case of deferred delivery, a reference to the bundle is linked to the registration. If the registration is set up for script execution and is passive, then the relevant script is run. If this results in the registration becoming active the bundle will be delivered to the activated registration.

An active registration will be bound to an application connection and the application can use the API to send and receive bundles using one of the interfaces defined in Sections 4.4.1and 4.4.2). If the application creates a new bundle, the Bundle Construction and Dissection Component is used to construct the bundle data structure (see item 130 and Section 4.6.2.5). It will be stored using the Bundle Storage component (Section 4.10) and the Bundle State management component, which records the Forwarding Log for the bundle (Section 4.6.6), when notified of the existence of a new bundle. Bundles that are received on a registration will be dissected and reformatted for transmission across the API using the Bundle Construction and Dissection Component (Section 4.6.2.3). In principle, multiple registrations can be bound to a connection at any time.

The BD also contains some ‘internal’ applications to which bundles may be delivered and which can also send bundles. The internal applications (Sections 4.12.1 to 4.12.7) register the appropriate EIDs to which bundles are delivered in accordance with their destination EID, including the *service tag*.

The DTN2 API contains some parameters intended to support a publish/subscribe mechanism that has not been fully described or documented. These functions are not required by the N4C applications at present and the functionality has not been extensively trialled. For compatibility the parameters are included and (so far as is possible) described, but the N4C implementation does not require these parameters to be supported.

4.4.1 Primary API (compatible with DTN2 at release 2.7.0)

This is the ‘classic’ API designed into the DTN2 reference implementation. Note that the descriptions here give all the parameters separately. If using the C++ binding in DTN2, many of the parameters are subsumed into structures and the parameters will be structure members. The Python binding, on the other hand, uses the parameters explicitly.

Whilst this is the commonly used API, it has a number of drawbacks and the Secondary API (Section 4.4.2) is intended to overcome these drawbacks. Problems that have been identified include the half-duplex send/receive mechanism, lack of asynchronous notification mechanism for arriving bundles and rudimentary capabilities for adding subsidiary blocks to the bundle.

This API is designed around a Sun RPC style interprocess communication channel. API parameters are encoded and decoded using the XDR paradigm and client-server communications use a TCP based socket connection that can work either intra- or inter-machine.

[The DTN2 code for this API is not very robust. In particular, the *handle* parameter is not checked for validity. Use of an invalid (e.g., closed) handle is likely to result in a fatal exception.]

4.4.1.1 Primary API Functions

4.4.1.1.1 dtn_open

Open a connection from the application to the Bundle Daemon (BD). If successful, returns a success code and an opaque handle, which is used to identify this connection in all other calls to this API. On error, returns an appropriate error code denoting the reason for failure. This function is always called first in order to get the communications handle. In DTN2, failing to initialize the handle is likely to lead to a fatal exception as the code is not protected against using an invalid or closed handle as an API parameter. The function opens a TCP stream socket which is used for all client-server communications, and initializes the RPC/XDR interface. By default the BD listens on port 5010 and is expected to be accessible via the loopback IP interface on the same machine as the application is running on. These defaults can be overridden by setting the environment variables DTNAPI_PORT and DTNAPI_ADDR for the client application process and for the BD process, or using the API configuration commands for the BD (see items 366 and 367).

Parameters:

Direction	**Name**	**Value Type**	**Description**
Out	Result	Enumerated: { DTN_SUCCESS, DTN_ECOMM. DTN_EVERSION }	Result code: Success or failure code. Failures result either from being unable to connect with the BD (DTN_COMM) or the application client API being a different version from the BD server API (DTN_EVERSION).
Out	Handle	Opaque	Session handle to be used with all other calls.

4.4.1.1.2 dtn_close

Terminate a connection to the BD. Should be called after completing use of the connection, but will normally be (effectively) called on application program exit.

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).

4.4.1.1.3 dtn_errno

Returns the success or error status relating to the last operation performed using the handle from the connection handle.

Parameters:

Directio n	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
Out	Errno	Enumerated: { DTN_SUCCESS, DTN_ECOMM. DTN_EVERSION, DTN_EINPOLL, DTN_EXDR DTN_EINVAL, DTN_ENOTFOUND. DTN_EBUSY, DTN_ENOSPACE, DTN_ETIMEOUT, DTN_EINTERNAL }	Error Codes from RPC sub-system: (Success) Application to server comms failure Client/server version mismatch Operation not allowed when polling XDR data corrupt when unpacking Invalid parameter value passed Registration, etc., not found Invalid operation: registration in use No (file) space for bundle (Unexpected) timeout from 'poll' Internal BD server error

4.4.1.1.4 dtn_set_errno

Force the connection handle error status. Affects only the client - there is no operation on the server and does not affect previous result values.

Parameters:

Directio n	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Errno	ENUMERATED (for possible values see Section 4.4.1.1.3.)	Force the error status of the session handle. Does not communicate with the server or

			do any other operations.

4.4.1.1.5 dtn_build_local_eid

Build an appropriate local endpoint id (EID) by appending a slash ('/') separator and the specified 'service tag' to the daemon's preferred administrative endpoint id. It assumes that the administrative endpoint id is taken from the dtn: URI scheme at present. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	
In/Out	Local_eid	ASCII String	Concatenation of BD's preferred administrative EID, a slash character and the *service tag* parameter. There is a fixed length limit on this string (currently 255 characters, but this is a bug - the limit should be length of scheme name + 1024 characters - typically 1027 characters for dtn: scheme.)
In	Service_tag	ASCII String	Demultiplexing tag for bundle delivery at destination.

4.4.1.1.6 dtn_register

Create a dtn registration. If the *init_passive* flag in the reginfo structure is true, the registration is initially in passive state, requiring a call to dtn_bind to activate it. Otherwise, the call to dtn_bind is unnecessary as the registration will be created in the active state and bound to the handle. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Direction	**Name**	**Value Type**	**Description**
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_eid	ASCII String	EID to be matched by destination EID in bundles to be delivered to connections bound to this registration. The EID may contain one or more wildcard characters (*) so that bundles with a range of Destination EIDs will match the registration. Note that it is **not** checked that the EID is *subsumeed* by a Local EID (see Section 4.4.1.1.5 and item 440). This means that a registration with a URI authority part that doesn't match a Local EID will never have any bundles delivered to it.

In	Registration_flags	Bit-Field[32] {
		DTN_REG_DROP: 0
		DTN_REG_DEFER: 1
		DTN_REG_EXEC: 0+1
		
		DTN_SESSION_CUSTODY: 2
		DTN_SESSION_PUBLISH: 3
		DTN_SESSION_SUBSCRIBE: 4
		}
			
			Drop bundles received if registration is not active at delivery time.
			Save bundles for deferred delivery if registration is passive at delivery time.
			Execute script and defer delivery if registration is passive at delivery time.
			
			[These flags are used for the session publish/subscribe support not used in N4C. Definitions TBD.]
			
			NOTE: The flags are not entirely independent bit flags. The

			DTN_REG_* are mutually exclusive and the first two bits are interpreted together as a two bit field. NOTE: The **DTN_REG_EXEC** option has significant security implications. An attacker could cause the daemon to execute an arbitrary script remotely. Care should be taken with this option. **It also implies that the BD should NOT be run with root privileges.** There should probably be a way of disabling this option.
In	Lifetime	Unsigned Integer	Lifetime of registrations in seconds
In	Init_passive	Boolean	If TRUE set registration passive initially. If FALSE set registration active and bind to the connection being used for this call.
In	Script	ASCII String (with length)	Script to be executed when a bundle arrives when registration is passive if Registration_flags include DTN_REG_EXEC. The string is passed to the *system* library call for execution by the shell command interpreter. Note that the process context in which this string is interpreted is that of the BD. Thus the command needs generally needs to include a full pathname. The resulting process will be owned by the same user that started the BD rather than the owner of the application process that performed the registration which may constrain the actions of the resulting process. The script should run to completion within a short period as otherwise it blocks the main bundle core thread.
Out	Result	ENUMERATED {	Result of operation: New Registration_id returned in next

		SUCCESS(0), FAIL(-1)}	parameter if Result is SUCCESS.
Out	Registation_id	Unsigned Integer	Identification of registration created.

4.4.1.1.7 dtn_unregister

Remove a dtn registration.

If the registration is in the passive state (i.e., not bound to any handle), it is immediately removed from the system. If it is in active state and bound to the given handle, the removal is deferred until the handle unbinds the registration or closes. This allows applications to pre-emptively call unregister so they don't leak registrations. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Direction	**Name**	**Value Type**	**Description**
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_id	Unsigned Integer	ID of Registration to be removed as previously returned by *dtn_register* or *dtn_find_registration*.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.8 dtn_find_registration

Check for an existing registration on the given endpoint id (EID), returning DTN_SUCCESS and filling in the registration id if it exists, or returning DTN_ENOTFOUND if it doesn't. If the call fails the error code can be retrieved with *dtn_errno*. Note that the check is **not** a check for an *exact* match, but for any registration, including ones with wildcards (*) which the given EID would be delivered to; there would also be a match if the given EID contains wildcard characters (*) and a registration would match this pattern. [This behaviour may not be desirable. It could lead to bundles being delivered to the application which it wasn't really expecting (if the matched registration is a pattern) or only getting a subset of the expected bundles (if the given EID is a pattern and the matching registration is more restricted). An exact string match might be more appropriate solution here. Also there is no way to check exactly what the matched registration's EID actually is and whether it contains wildcards.]

Parameters:

Direction	**Name**	**Value Type**	**Description**
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see

			Section 4.4.1.1.1).
In	Endpoint_id	ASCII String	Check for registration that this EID matches.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation: New Registration_id returned in next parameter if Result is SUCCESS.
Out	Registation_id	Unsigned Integer	Identification of registration found.

4.4.1.1.9 dtn_change_registration

Modify an existing registration. (Intended) functionality similar to *dtn_register* but modifying the existing registration.

[TBD: This function is not implemented in DTN2. The practical value of this function is unclear - it might be that altering the lifetime or the script associated with an existing registration for a given EID might be useful as it would ensure that any pending bundles are not dropped as would happen if *dtn_unregister* followed by *dtn_register* had to be used. Changing the EID or the flags is not useful and *init_passive* applies to the initial state.]

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_id	Unsigned Integer	ID of Registration to be modified as previously returned by *dtn_register* or *dtn_find_registration*.
In	Registration_eid	ASCII String	EID to be matched by destination EID in bundles to be delivered to connections bound to this registration.

In	Registration_flags	Bit-Field[32] {
		DTN_REG_
		DROP: 0
		DTN_REG_
		DEFER: 1
		DTN_REG_
		EXEC: 0+1
		
		DTN_SESSION_
		CUSTODY: 2
		DTN_SESSION_
		PUBLISH: 3
		DTN_SESSION_
		SUBSCRIBE: 4
		}
			Drop bundles received if registration is not active at delivery time.
			Save bundles for deferred delivery if registration is passive at delivery time.
			Execute script and defer delivery if registration is passive at delivery time.
			
			[These flags are used for the session publish/subscribe support not used in N4C. Definitions TBD.]
			
			NOTE: The flags are not entirely independent bit flags. The DTN_REG_* are mutually exclusive and the first two bits are interpreted together as a two bit field.

In	Lifetime	Unsigned Integer	Lifetime of registrations in seconds
In	Init_passive	Boolean	If TRUE set registration passive initially. If FALSE set registration active and bind to the connection being used for this call.
In	Script	ASCII String (with length)	Script to be executed when a bundle arrives when registration is passive if Registration_flags include DTN_REG_EXEC.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation. Currently always returns FAIL as it is not implemented.

4.4.1.1.10 dtn_bind

Associate a registration with the current server communication channel. This serves to put the registration in 'active' mode. Note that this function is called implicitly if *dtn_register* (see Section 4.4.1.1.6) is called with *init_passive* set to FALSE. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Directio n	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_id	Unsigned Integer	ID of Registration to be bound as previously returned by *dtn_register* or *dtn_find_registration*.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.11 dtn_unbind

Explicitly remove an association from the current server communication handle. Note that this is also implicitly done when the handle is closed. This serves to put the registration back into 'passive' mode. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Directio n	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_id	Unsigned Integer	ID of Registration to be unbound as previously returned by *dtn_register* or *dtn_find_registration*.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.12 dtn_send

Send a bundle either from memory or from a file. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Directio n	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open* (see Section 4.4.1.1.1).
In	Registration_id	Unsigned Integer	If the bundle is to be sent as part of a SESSION, this parameter contains the ID of Registration associated with the session as previously returned by *dtn_register* or *dtn_find_registration*. If the bundle is sent 'independently' this value can be

			DTN_REGID_NONE.
In	Source_EID	ASCII String	The EID to be sent as the 'Source EID' of the bundle. The EID must be a valid URI but is not otherwise checked[1]. The bundle cannot request reports or custody transfers, and must not be fragmented if the source is the NULL EID.
In	Destination_EID	ASCII String	The EID to be sent as the 'Destination EID' of the bundle. The EID must be a valid URI but is not otherwise checked.
In	Report_EID	ASCII String	The EID to be sent as the 'Replyto EID' of the bundle. If this string is empty ("") then the NULL EID is used.
In	Priority	Enumerated: { COS_BULK, COS_NORMAL. COS_EXPEDITED, COS_RESERVED}	Delivery priority for bundle: - lowest priority - regular priority - important - TBD
In	Delivery_Optio ns	Bit Field [32]: { DOPTS_NONE: 0, DOPTS_CUSTODY: 1, DOPTS_ DELIVERY_RCPT: 2, DOPTS_RECEIVE_RCPT: 3, DOPTS_ FORWARD_RCPT: 4, DOPTS_ CUSTODY_RCPT: 5, DOPTS_	Various delivery option bits: - no delivery options wanted - custody transfer requested - end to end delivery (i.e. return receipt) requested - per hop arrival receipt requested - per hop departure

[1] [In DTN2 the comments and documentation state that the source EID must correspond to an existing registration in the BD. This does not make any sense and the code no longer implements this constraint in the API. However the constraint still implemented in the control interface *bundle inject* command (see item 368) and the *bundle injects* event (see item 64). This is probably correct as injected bundles ought to originate at the local node and should probably be associated with some registration that receives bundles.]

		DELETE_RCPT: 6, DOPTS_ SINGLETON_DEST: 7, DOPTS_ MULTINODE_DEST: 8, DOPTS_ DO_NOT_ FRAGMENT: 9 }	receipt - per custodian receipt - request deletion receipt - destination is a singleton - destination is not singleton - set the do not fragment bit
In	Lifetime	Unsigned Integer	Lifetime in seconds. The server will set the expiration time to be the server current time plus the lifetime.

In	Payload_type	Enumerated: {
		DTN_PAYLOAD_FILE,
		DTN_PAYLOAD_MEM,
		DTN_PAYLOAD_TEMP_
		
		FILE
		}
			Specify the way in which the payload data is transferred to the server. The payload of a bundle will normally be stored in a file. If the payload is transferred in MEMory the data is written to the file. If via a FILE, the data is copied to the bundle payload file. If TEMP_FILE the file specified is renamed to be used as the bundle payload file if possible.

In	Sequence_ID	ASCII String	[Only used in Publish/Subscribe functions. Assign as “” (empty string) if not in use.]
In	Obsoletes_ID	ASCII String	[Only used in Publish/Subscribe functions. Assign as “” (empty string) if not in

			use.]
In	Ext_Blk_Count	Unsigned Integer	Number of extension blocks attached in next parameter.
In	Ext_Blk_Data	Array of extension blocks	Array of extension block specifiers, containing the block type, block flags, block data length and block data for each block.
In	Meta_Blk_Count	Unsigned Integer	Number of metadata blocks attached in next parameter.
In	Meta_Blk_Data	Array of extension blocks	Array of metadata block specifiers, containing the block type, block flags, block data length and block data for each block.
In	Payload	Structured buffer containing: - Payload type - Filename length and pointer to string - Memory buffer length and pointer to data buffer - Pointer to status report structure	Either the name of the file containing the payload or the actual payload depending on the Payload_type parameter. The status report field is only used on reception.
Out	Sent_Bundle_ID	Structure containing two 64 bit unsigned integers: - Creation timestamp - Sequence number in creator BD.	Identifier for bundle sent.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.13 dtn_cancel

Cancel a bundle transmission. If the call fails the error code can be retrieved with *dtn_errno*.

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open*.
In	Sent_Bundle_ID	Structure containing two 64 bit unsigned integers: - Creation timestamp - Sequence number in creator BD.	Identifier for bundle to be cancelled.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.14 dtn_recv

Blocking receive for a bundle, filling in the spec and payload structures with the bundle data. The location parameter indicates the manner by which the caller wants to receive payload data (i.e., either in memory or in a file). The timeout parameter specifies an interval in milliseconds to block on the server-side (-1 means infinite wait). If the call fails the error code can be retrieved with *dtn_errno*. If the call returned because the timeout expired before a relevant bundle was received, the call will fail with error code DTN_ETIMEOUT.

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open*.

In	Payload_type	Enumerated: {
		DTN_PAYLOAD_FILE,
		DTN_PAYLOAD_MEM,
		DTN_PAYLOAD_TEMP_FILE
		}
			The payload can either be requested to be returned as a file or in a memory buffer. For received bundles there is no difference between _FILE and _TEMP_FILE. The server decides on the file location and name. If the payload exceeds a server defined threshold, the payload will be returned as a file even if requested in memory.

In	Timeout	UNIX style timeval (secs/ms)	Length of time in milliseconds to block waiting for a bundle.
Out	Bundle_Spec	Bundle_spec_type (see below)	Information about the bundle similar to the parameters needed for *dtn_send*.
In/Out	Payload	Structured buffer containing: Payload type Filename length and pointer to string Memory buffer length and pointer to data buffer Pointer to status report structure	On input, null (just to give the space for output) On output the payload file name or memory buffer with the payload depending on the payload type. If the payload has been detected to be a status report, it is decoded and a status report structure returned in addition.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

The *Bundle_spec_type* contains the following information about the bundle:

- Source EID
- Destination EID
- EID to which reports should be sent
- Priority code
- Delivery options flags
- Expiration time
- Creation time
- The next three items are only relevant for bundles associated with publish/subscribe sessions:
  - The registration id of the session with which the bundle is associated
  - The sequence id of the bundle in the session
  - The sequence id of a bundle which this bundle obsoletes in the session.
- The count of blocks in the extension block list followed by the blocks
- The count of blocks in the metadata block list followed by the blocks

The types of these items match the types used in *dtn_send* (Section 4.4.1.1.12).

The extension and metadata blocks each contain:

- The block type
- A set of extension block flags

- The length of the block body in octets
- The block body as an opaque unit

If the payload is a status report, the payload report status points to a structure that contains:

- The bundle id that the report is about.
- The reason code for the status report from the following enumeration:
  - REASON_NO_ADDTL_INFO = 0x00,
  - REASON_LIFETIME_EXPIRED = 0x01,
  - REASON_FORWARDED_UNIDIR_LINK = 0x02,
  - REASON_TRANSMISSION_CANCELLED = 0x03,
  - REASON_DEPLETED_STORAGE = 0x04,
  - REASON_ENDPOINT_ID_UNINTELLIGIBLE = 0x05,
  - REASON_NO_ROUTE_TO_DEST = 0x06,
  - REASON_NO_TIMELY_CONTACT = 0x07,
  - REASON_BLOCK_UNINTELLIGIBLE = 0x08

- A set of report flags showing which timestamps are valid:
  - STATUS_RECEIVED = 0x01,
  - STATUS_CUSTODY_ACCEPTED = 0x02,
  - STATUS_FORWARDED = 0x04,
  - STATUS_DELIVERED = 0x08,
  - STATUS_DELETED = 0x10,
  - STATUS_ACKED_BY_APP = 0x20,

- A set of timestamps, only one of which will normally be filled in for a given status report
  - Receipt timestamp
  - Custody timestamp
  - Forwarding timestamp
  - Delivery timestamp
  - Deletion timestamp
  - Acknowledged by application timestamp [Not implemented by DTN2.]

4.4.1.1.15 dtn_session_update

Blocking query for new subscribers on a session. One or more registrations must have been bound to the handle with the SESSION_CUSTODY flag set. Returns an indication that the subscription state in the given session has changed (i.e., a subscriber was added or removed). If the call fails the error code can be retrieved with *dtn_errno*. If the call returned because the timeout expired before a relevant bundle was received, the call will fail with error code DTN_ETIMEOUT. [This function is not currently required by N4C.]

Parameters:

Direction	**Name**	**Value Type**	**Description**
In	Handle	Opaque	Session handle previously

			returned by *dtn_open*.
In	Timeout	UNIX style timeval (secs/ms)	Length of time in milliseconds to block waiting for a notification.
Out	Status	Bit array[32]={ DTN_SESSION_ SUBSCRIBE}	At present only SUBSCRIBE sessions are flagged here. Otherwise this is 0.
Out	Session EID	ASCII String	Session EID from session.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.1.1.16 dtn_poll_fd

Return a platform operating system file descriptor for the given handle suitable for use with *poll()* or *select()* system calls in conjunction with a call to dtn_begin_poll(). [This call does not access the server. In DTN2 the handle is not checked for validity and the call may either provoke a process exception (core dump due to invalid pointer access) or return am invalid file descriptor. It should only be called on a valid, open handle. It should be possible to improve on this and return (-1) if the handle is not open or is otherwise invalid.]

Parameters:

Direction	Name	Value Type	Description
In	Handle	Opaque	Session handle previously returned by *dtn_open*.
Out	Result	File descriptor (Integer)	(Socket) file descriptor on which handle is exchanging data with server.

4.4.1.1.17 dtn_begin_poll

Begin a polling period for incoming bundles. Returns a file descriptor suitable for use with *poll()* or *select()* (similar to *dtn_poll_fd* - Section 4.4.1.1.16). Note that *dtn_bind()* (Section 4.4.1.1.10) must have been previously called at least once on the handle. The client side of this function is non-blocking, i.e., it does not wait for a response from the server after sending the request message to the server.

If the kernel returns an indication that there is data ready on the file descriptor, a call to *dtn_recv* will then return the bundle without blocking. Thereafter *dtn_begin_poll* must be called again to wait for more bundles.

Also, no other API calls besides *dtn_recv* (Section 4.4.1.1.14) can be executed during a polling period, so an application must call *dtn_cancel_poll* (Section 4.4.1.1.18) before trying to run other API calls. Other calls will elicit the error code DTN_EBUSY if called during a poll period.

The server handler for *dtn_begin_poll* sends a response either when a bundle is received or when the poll times out. This response is expected either in *dtn_recv* or *dtn_cancel_poll* and is read before the response to the message resulting from the second call if the handle had recorded that it was in the polling state.

Parameters:

Direction	Name	Value Type	Description

In	Handle	Opaque	Session handle previously returned by *dtn_open*.
In	Timeout	UNIX style timeval (secs/ms)	Length of time in milliseconds to remain in polling state.
Out	Result	File descriptor (Integer)	(Socket) file descriptor on which handle is exchanging data with server. If the sending of the *dtn_begin_poll* message to the server fails the return value will be (-1) which is an invalid file descriptor.

4.4.1.1.18 dtn_cancel_poll

Cancel a polling interval. This function should only be called when the connection is in the polling state started by calling *dtn_begin_poll* (Section 4.4.1.1.17). If the call fails the error code can be retrieved with *dtn_errno*. The error code will be DTN_ETIMEDOUT if the polling period set by *dtn_begin_poll* had expired.

Parameters:

Direction	**Name**	**Value Type**	**Description**
In	Handle	Opaque	Session handle previously returned by *dtn_open*.
Out	Result	ENUMERATED { SUCCESS(0), FAIL(-1)}	Result of operation.

4.4.2 Secondary API (based on D-Bus)

This will be specified at a later time.

4.4.3 Registration Management Functionality

Registrations represent an interest by some current or potential application in bundles sent to this BD. As explained in Section 4.4, registrations are maintained in persistent storage indexed by a serial number (*registration id*) that is unique for the lifetime of the persistent store and recreated if the BD is restarted. At any time a registration can be in an *active* or a *passive* state, i.e., there is an active application willing to accept bundles for the local EID modified by a specified service tag, or currently no application is willing to accept such bundles. All registrations have a lifetime after which they will be deleted, but can be refreshed by an application if desired. Bundles addressed to a passive registration can either be held for later delivery (deferred) or deleted. Deletion may be immediate if the BD is configured to allow *early deletion* of bundles (see item 34) or otherwise as and

when the bundle's lifetime expires, but note that the bundle for which delivery to a passive registration has been attempted once will not be reattempted even if the registration becomes active before the bundle expires.. A bundle arriving at the BD with a local EID as destination but with a service tag for which there is no registration will be deleted (either immediately if early deletion is enabled or when it expires).

Registrations maintain a *delivery cache* of (recently) delivered bundles for each registration in order to monitor and suppress delivery of duplicates, indexed by *Global Bundle Or Fragment Identifier* (*GBOFID).*

Registration management maintains a table of all current registrations for the BD.

A number of *registration ids* with low values are reserved for internal applications and the allocation of other registration values with the value after the last reserved *registration id* (currently 9). The following registrations are reserved:

- Internal administrative application (see Section 4.12.5).
- Link state router registration [Not now used in DTN2 - ?superseded by DTLSR registration.]
- Ping echoer internal application (see Section 4.12.1)
- Management by bundle application (see Section 4.12.3). [Not implemented in DTN2.]
- External router application (see Section 4.12.7) [Not fully documented as not needed for N4C]
- Delay Tolerant Link State Router (DTLSR) Link State Advertisement receiver application. (see Section 4.12.6)[Not fully documented as not needed for N4C.]
- Internal Heartbeat application (see Section 4.12.4) [Not implemented in DTN2.]

The following functionality is provided to handle registrations:

17. **Maintain a serial number counter for the registrations** that is used to allocate a unique ID for each registration created. The last value used is stored in persistent storage (see Section 4.10.3).

18. **Create a *registration* for a given local EID and *service tag***. Allocate a unique ID for the registration (must be unique even if the BD is restarted until the persistent store is reinitialized). Start an *expiration timer* for the registration. Record the state of the registration (passive/active; delivery failure action - defer, delete or execute script; name of script to execute if specified). Store in persistent storage. Create an empty *delivery cache*. Add it to the table of registrations. Note that duplicates of bundles already recorded in the *delivery cache* are *never* delivered to the *registration*; unlike the forwarding decision made by routers where the BD can be configured to forward duplicate bundles, there is no ability to have duplicates (selectively) delivered to *registrations* even if the application was interested. This position could be modified relatively easily if there was any particular need for delivery of duplicates.

19. **Delete a *registration* using its unique ID**. Clear the cache of delivered bundle identifiers. Delete the registration from the table of registrations and persistent storage.

20. **Deliver bundles to the application using a *registration***. This may be an internal application or an application registered via the API. The bundle identification information will be added to a cache of information about recently delivered bundles (the *delivery cache)* for this registration unless this bundle is a duplicate of a bundle that has been delivered to this registration. The delivery of second and subsequent copies will be suppressed. [DTN2 does not apparently manage this cache so it could grow without limit for a long lived registration. It should probably be cleared out of the reception cache when a bundle reaches its expiry time.]

21. **Find *registrations* matching a given EID which may include a service tag**. Matching is done according to the rules of the EID scheme to which the EID belongs (see Section 4.7.1 and item 180.3).

22. **Manage an expiration timer for each *registration*.** Set the timer when the *registration* is created and whenever it is refreshed through the API (see items 4.4.1.1.6 and 4.4.1.1.9). When the timer expires post a *Registration Expired* event on the bundle core.

4.4.4 Signals Sent to Other Components

23. When a new *registration* is created by a client a *Registration Added* event is sent to the Bundle Daemon Core (see item 72).

24. When a client unregisters a *registration*, a *Registration Removed* event is sent to the Bundle Daemon Core unless the registration is bound to the client (see item 73), in which case the event is not sent until the client is closed or the registration is explicitly unbound. It is also not removed if some other client has the registration bound. Delete any lifetime timer associated with the *registration*.

25. When a client unregisters a *registration*, the registrations bound to the client are examined. If any have expired, they are removed by sending a *Registration Expired* event to the Bundle daemon core (see item 74).

26. When a client is closed, the *registrations* bound to the client are examined. If any have expired, they are removed by sending a *Registration Expired* event to the Bundle Daemon core (see item 74).

27. When a bundle is received by the API, a *Bundle Accept Request* event is sent to the Router to verify that there is adequate storage available to hold the persistent version of the bundle (see item 67).

28. When a bundle is received by the API (*dtn_send* function), a *Bundle Received* event is sent to the Bundle Daemon Core (see item 56).

29. When a received bundle has been successfully passed to a client, a *Bundle Delivered* event is sent to the Bundle Daemon Core (see item 58).

4.4.5 Functionality Invoked by Signals Sent to this Component

No events are sent to this component from the Bundle Daemon Core. Its functionality is only invoked by API clients calling API functions.

4.5 BUNDLE DAEMON CORE

The **Bundle Daemon Core** (or *bundle core,* for short) is the master component of the BD that manages

- controlling the network interfaces that DTN in the local node uses,
- sending and receiving DTN node advertisements on interfaces that have been configured to handle DTN node discovery,
- opening and closing links to other nodes either configured through the control interface or as a result of discovery advertisements,
- the router(s) that determine and schedule bundles for transmission on the network interfaces,
- sending and receiving bundles over the network interfaces,
- storing and retrieving bundles from persistent storage in the local node,
- accepting bundles from and delivering bundles to applications in the local node in line with registrations, and
- managing timers needed for various purposes including bundle expiry, custody retransmission, link idle shutdown and, optionally, BD idle shutdown.

Once the Bundle Daemon Core thread has been initialized, it acts as an event dispatcher that drives the remainder of the components that handle these functions. Various subsidiary threads may be created and destroyed to manage functions (especially those associated with communications endpoints) that need to run asynchronously.

4.5.1 Core Start-up and Shutdown

During start-up of the Bundle Daemon (BD) core

30. **A default NULL Local EID (*dtn:null*) is assigned to the node**.

31. **An empty event queue is created**. Provision is made for notification to be made when the queue is empty to allow for the BD to shut itself down if the queue remains empty for s significant time.

32. **Lists and queues are created to hold information about in memory bundles**. These include the *all bundles* list (a complete list of all extant bundles), *pending bundles* queue (those bundles awaiting transmission or delivery), and *custody bundles* list (those bundles for which the node has accepted custody).

33. **Other component instances are created and initialized for the Contact Manager, the Fragmentation Manager and the router(s)**.

34. **A number of configurable parameters are initialized to enable or disable certain features in the BD**. These are:

    - **Allow early deletion of bundles**: Bundles which are not apparently needed further can be deleted before the expiry time specified in the bundle with the consent of the router in use (defaults to true).
    - **Suppress processing of duplicate copies of bundles received**. [DTN2 has a rather complex 'tree' of duplicate suppression mechanisms. Only this top level mechanism is configurable.] Received bundles that are duplicates of bundles

already in the *pending queue* are not processed further after identification in the bundle core *bundle received* event handler if this flag is true (defaults to true). If the duplicate is not suppressed at this stage, the bundle will be offered for delivery to any *registration* that its destination EID matches and offered for forwarding through the currently active *bundle router*. However duplicates will *never* be delivered to *registrations* irrespective of the setting of this flag (see items 18 and 20). For *Table-based* Routers, if the bundle is already in the router's *reception cache*, it will not be forwarded (see Section 4.8.2) but this functionality is not applicable to all routers.

- **Accept custody of bundles**. Take custody of received bundles that request custody management. (defaults to true).
- **Enable reactive fragmentation**. When using a reliable convergence layer with acknowledgements, bundles that have been partially sent or received can be turned into fragments if the link contact breaks during communication of the bundle (defaults to true).
- **Retry transmissions of bundles on links using a reliable convergence layer that are not acknowledged at all**. (i.e., some octets have been sent but none have been acknowledged).
- **Store injected bundles only in memory** (not in persistent data store). This is an optimisation that is used by the external router only at present. [Not relevant for N4C] (defaults to false).
- **Test permuted delivery**. This is a testing capability provided for DTN2 that allows the API to deliver bundles to an application in a random order (defaults to false).

35. **If installed, the Bundle Security Protocol component cipher suites are initialized** (see Section 4.6.3).

When the Bundle Daemon Core thread commences running

36. **Existing registrations are loaded into memory from persistent storage** . Additional registrations are added for the internal applications (see Section 4.12). Note that some registrations may have expired while the BD was not running and will need to be deleted. A *Registration Added* Event is posted for each unexpired registration quoting the source of the event as STORE (see item 72).

37. **Any bundles in persistent storage are loaded into memory** During this process it is checked that all information is available for the bundle, deleting any bundles which cannot be completely reloaded. Note that some bundles may have expired during the time that the BD was shutdown and will need to be deleted. A *Bundle Received* Event is posted for each valid, unexpired bundle quoting the source of the event as STORE (see item 56). [In principle, reloading a bundle should include reloading its forwarding log which records what has been done in terms of transmission of the bundle. DTN2 does not currently write the forwarding log to persistent store. This means that the *Bundle Received* Event will have to treat these bundles as newly received. This may result in some duplicate transmission.] The reloaded bundles will be referenced on the appropriate lists as they are reloaded. Some post-processing of blocks may have to be done for certain types of blocks (especially security blocks) after reloading.

38. **The timer subsystem is started**. Timer events arise from the following situations:
    - **A bundle's lifetime has expired**; the bundle can be deleted.
    - **A registration's lifetime has expired**; the registration can be deleted when it is next inactive.
    - **The custody timer for a bundle for which this node has custody has expired**; the bundle may need to be retransmitted.
    - **If the BD is configured to terminate after a period of idleness, a timer event occurs when the idle period has expired**; the BD exits gracefully.
    - **For links defined as ALWAYS_ON and ON_DEMAND, when the link availability timer expires; this timer is started when one of these links closes unexpectedly rather than on user request or because of being idle**. The contact manager attempts to reopen the link, using exponential back off of time between retries to avoid overload. [Note DTN2 does not use timer timeouts to manage link idle timeouts with ON_DEMAND links because there are already timed events going on (poll timeouts) that allow the link to determine when it has been idle for too long.]
    - **The DTN2 Ethernet convergence layer periodically sends beacons to support neighbour discovery**.
    - **The DTLSR Router uses timeouts to schedule periodic sending of LSAs**.
    - **The External Router interface uses timeouts to schedule periodic sending of Hello messages**.
    - **The PRoPHET Router uses timeouts to schedule metadata link keepalive messages**.
    - **The Table-based Routers use timeouts to cancel bundle transmissions on links that have gone down and do not appear to be about to reopen**.
    - **The Table-based Routers use timeouts to schedule periodic refreshes for subscriptions**; this involves sending a subscription request bundle generated within the router.

39. **The time of the last event is initialized to the current time**.

During graceful shutdown

40. **Other components are notified of shutdown and allowed to terminate any in progress operations before the instances are deleted**. Note that the persistent store is intended to be updated whenever a change is made an the associated in-memory object, so that it should not be necessary to do any updates to the database before closing any connections and terminating the application. This means that unexpected shutdowns should not corrupt the persistent data, provided that a suitably robust database package is chosen. [DTN2 has special facilities for leaving a marker *clean shutdown* file if the Berkeley database was shutdown cleanly rather than as a result of an unexpected shutdown.]

4.5.2 Core Functionality

During the operation of the Bundle Daemon Core the following functionality is provided:

41. **Statistics of number of bundles processed in various categories is maintained**. Categories are received, delivered, generated, transmitted, expired, deleted,

duplicates received, injected. [It might be useful to keep track of the number of octets processed in each category.]

42. **A count of the number of events processed is maintained**.

43. **The statistics and event count can be accessed by a control interface command** (see Section 4.11.7.2).

44. **The time at which the most recent event processing started is recorded**. A warning message is logged if there is a significant delay between messages.

45. **If configured by the user, the BD can terminate gracefully if it remains idle** (no events to process) for longer than a period configured through the control interface (see item 382).

4.5.3 Event Distribution and Event Handlers

The Bundle Daemon Core operates as an event dispatcher for the rest of the system. In the DTN2 implementation it is not technically a finite state machine because all events can be handled at any time. A single event queue is maintained which is serviced by the master dispatch routine. Events can be posted either at the head of the queue (LIFO) or, in most cases, at the tail of the queue. In either case, if the event is posted from another thread, the posting thread can be made to wait on completion of event processing and be notified when this happens to restart the posting thread. Care should be taken to ensure that all event handlers are non-blocking and do not take an excessive time to run to completion. The event handler runs as a continuous loop that does the following

46. **Checks if BD termination has been requested**. If so, exits the loop and shuts down the BD.

47. **Checks the timer subsystem to see if any timers have expired**. Runs code to handle the resultant timer events and returns the time interval before the next timer event will occur (unless a new timer is scheduled).

48. **If there are any events in the event queue, pop the event at the head of the queue and process it**.

49. **If the event was apparently posted after the current time log a warning** (time apparently ran backwards!)

50. **If the event was in the queue for an excessive amount of time, log a warning**.

51. **Dispatch the event to the local Bundle Daemon Core event handlers**. The Bundle Daemon Core event handlers may determine that no further processing is required, and set a flag that suppresses any additional processing.

52. **Unless suppressed by the Core processing, send the event to the router event handler and the contact manager event handler**.

53. **On completion of all handlers, monitor the length of time taken to process the event and report a warning if the time seems excessive**. Record the current time as the time of the last event processed.

54. **On completion of all handlers, if the event related to a specific bundle, check if the bundle is required for future processing**. If not, delete it if early deletion is allowed (see item 34). The router may explicitly request that a bundle is retained even if it would otherwise be deleted by early deletion. If custody is no longer required it should be removed. Deleted bundles are removed from the lists of bundles as appropriate.

55. **If a bundle is deleted and either the local node had custody of the bundle or the bundle specifically requested deletion reports, send a *deletion report status report* administrative bundle (see item 121)**.

4.5.4 Event Handlers

The Bundle Daemon Core processes all the events in the system. There are a large number of event types. The following table summarises the functionality for each event type. In the case of the various bundle handling events, the data structure passed to the event handler will contain a reference to the affected bundle so that event handler routines can access the bundle easily. Similarly where relevant the event handler routines will be passed a reference to the link associated with the event (e.g., where a link is opened) and/or the registration associated with an event.

A significant number of these events (the query/report event pairs especially) are only used by the External Router and External Convergence Layer components in DTN2. These components are implemented in separate processes that communicate with the main BD via an Inter-Process Communication interface where the parameters of messages are expressed in an XML schema. This interface and the associated components are not used in N4C. The XML-encoded interface allows routing protocols and convergence layers to be prototyped in a convenient way, but the verbosity of the XML encoding and the context switching overhead involved in the large number of message exchanges, especially for routing, makes these components a questionable choice for a production environment. Using dynamically linked libraries might be a preferable implementation strategy for an extensible router/convergence layer architecture.

56.	bundle received	Process an arriving bundle. Incoming bundles can stem from various sources and what happens is dependent on the source. The source maybe any of: • PEER - the bundle has been received over the network from another node • APP - the bundle was created by a local application (note that bundles from internal applications in DTN2 may use ADMIN source - especially for the ping echoer application. • STORE - the bundle is being read in from persistent store at daemon startup • ADMIN - the bundle has been created to fulfill an

administrative requirement (e.g., a status report bundle)
- FRAGMENTATION - the bundle was created by the Fragmentation Manager, e.g., as a result of a successful reassembly.
- ROUTER - the bundle was created by a router (e.g., to carry metadata to another node).

A number of things have to be done for each incoming bundle:
- Update statistics according to the source of the bundle (*received*, *generated* or none in the case of bundles read back from store).
- If the bundle was received from a PEER or generated by an Application update the forwarding log for the bundle to record the arrival.
- Record the EID of the previous hop if the bundle was received from a PEER and the EID can be deduced either from a *previous hop block* or the information from the link on which it was received.
- Do reactive fragmentation on incomplete bundles.
- If the bundle was received from a PEER, validate the bundle (see Section 4.6.2.2).
- Send reports if needed (see item 121).
- Check for duplicates. Suppress them if permitted by the parameters set for the BD (see item 34) If the duplicate is suppressed, the *bundle received* event is not passed on to the router or contact manager. However if we have previously accepted custody of this same bundle, a *custody signal* is sent to the previous custodian reporting a redundant reception (see item 97 for information about what happens when this *custody signal* is received at the previous custodian). Update the statistics of duplicates received.
- Make sure the bundle can be stored (see *bundle accept* event and router functionality item 183).
- If either the bundle failed to validate or the router reports that it is unwilling to accept the bundle, then delete the bundle (see item 119), suppress further distribution of this event and do no more in this handler.
- Add to *pending bundles* queue (see item 113).
- Store the new bundle in persistent storage unless the bundle is being read back in from persistent storage.
- Take custody if specified in bundle and the BD is configured to accept custody (see items 34 and

125).

- If this is a complete bundle, obsolete any bundle fragments in store that are part of this bundle.
- If this is a complete bundle, check if it should be delivered locally and do so if possible (see item 116).
- If it is a fragment, check if it should be delivered locally. If so then check if we now have a covering set of fragments for the bundle so that it can be reassembled. If so, reassemble the bundle and repost the *Bundle Received* event for the newly completed bundle (see item 171).
- If the bundle was found to be valid and acceptable to the router, then the event is processed by the current *router* (see item 187).

57. bundle transmitted

At least some of a bundle has been transmitted on a link in accordance with the directions of the router. An attempt at transmission has been made and the transmission may have been totally successful, or unsuccessful either partially or completely e.g., because the link may have broken before the whole bundle has been transmitted or the receiver has run out of space.

- Check that the bundle was really being transmitted (race conditions can result in redundant events here).
- Update the relevant link's transmitted statistics (see Section 4.9.4).
- Delete the bundle from the transmission link's *in flight* queue, check that it was not still on the link's to be sent queue (*send queue*) - it should have been removed when transmission started.
- Update the bundle forwarding log to indicate the bundle has been transmitted on the specified link.
- If the link is reliable and none of the bundle was reliably sent and the BD is configured to retry the transmission of completely unacknowledged bundles (parameter *retry_reliable_unacked*), update the forwarding log last entry to indicate TRANSMIT_FAILED. Later when the router scans the *pending list* this bundle will be rerouted and transmission attempted again.
- If the bundle was not completely sent and reactive fragmentation is allowed (parameter *reactive_frag_enabled*), attempt to convert the bundle into transmitted and untransmitted fragments. The break is put either at the point to

which acknowledgements had been received (for reliable link) and otherwise at the point up to which data had been sent. [I didn't think we did reactive fragmentation for unreliable links?] See item 168.

- Delete the wire format blocks generated in preparation for this transmission (see item 152).
- Generate a forwarding status report if requested (see item 121).
- Schedule *custody timer* to review the need for retransmission if we have custody of this bundle on this node.

58.	bundle delivered	A bundle has been successfully delivered to a local application. • Generate delivery status report if requested (see item 121). • Either release custody if we have it locally (see item 126) or send a custody signal to current custodian to confirm delivery (see item 123).
59.	bundle expired	Generated by the timer subsystem when a bundle has reached its expiry time. • Call *delete_bundle* giving LIFETME_EXPIRED reason (see item 119).
60.	bundle free	A bundle is no longer required. This event is automatically posted when the number of references to a bundle drops to one (i.e., it is only left in the *all bundles* list (see item 32). This is usually at the point where it is deleted from the *pending bundles* queue (see item 120). • Handle deletion of bundle from persistent datastore if it has been stored in persistent storage and finally dispose of the bundle object.
61.	bundle send	A bundle is to be transmitted on a specific link in accordance with the instructions of the router. • Queues the bundle on the specified link (see item 114). [In DTN2 this uses the *queue_bundle* utility function in bundle_actions]

62.	bundle cancel	An application or the command interface has requested that transmission of a bundle previously requested should be aborted. • If the bundle is associated with a link delete it from the *send queue* and/or *in flight* queue if possible and delete the bundle (see item 115). [In DTN2 this uses the *cancel_bundle* utility in Bundle Actions to cancel the bundle.]; otherwise just delete the bundle.
63.	bundle send cancelled	Deal with the aftermath of cancellation of a bundle from a link. • Log a warning if the bundle is still on the *send queue* or *inflight* queue of the link (it should have been removed because of the cancellation). • Delete wire formatted blocks, log the cancellation and update the forwarding log and add to the *cancelled bundles* statistics.
64.	bundle inject	Event generated to handle a bundle that is created locally but need not be placed into persistent storage. It is not clear that there is any use for this functionality at present. [It is currently only (potentially) used by the External Router capability in DTN2; I am not clear if any actual routers use it. Note also that the event is not recorded in the bundle's forwarding log (unlike bundle received case.) A comment indicates that it ought to be used in the DTLSR router for link state bundles.] • Check the bundle is locally generated, i.e., the source EID is the NULL EID (*dtn:null*) or is a EID with a current *registration* at this node. Please note the footnote about this constraint associated with the *dtn_send* API function in Section 4.4.1.1.12. • Build the bundle from supplied parameters, setting the expiry time to 5 minutes. • If not a fragment or created by the router, check if it is for local delivery and deliver it. • Place on the *pending bundles* queue (see item 113). • Store the bundle in persistent storage if the BD is not configured to store injected bundles only in memory (see item 34). • Update the statistics of *bundles injected* (see item 41). • Post a *bundle injected* event (see item 65).

65.	bundle injected	Event generated at the end of the bundle inject event handler. [Currently does nothing in DTN2 apart from in the External Router capability where the event is forwarded to the external code.. Would allow external code to verify completion of the injection?]
66.	bundle delete	Event that causes unconditional deletion of a bundle that is no longer required. • Delete bundle from all lists and remove from persistent storage (see item 119).
67.	bundle accept	Event generated by the API server when a new bundle to be transmitted has arrived via the API. Actioned by the router component which is aware of the resources available (especially storage) to handle this bundle (see item 183). The bundle will be rejected if the router returns that it cannot be accepted. (This check is done 'on the fly' rather than in the *bundle received* event, as the router is running on the same thread whereas the API needs to post the event and wait for the answer.) • The function checks to see if the bundle should/can be accepted, e.g., because of resource constraints.
68.	bundle query	Event generated by External Routers to request a list of pending bundles. [It appears DTN2 does not use this event apart from forwarding from the External Router interface. Not needed for N4C.] • Post a *bundle report* event (see item 69).
69.	bundle report	Event used to return a list of *pending bundles* to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the bundle list returned.] • No action required by Bundle Daemon Core. • Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.
70.	bundle attributes query	Event generated by External Routers to request a list the attributes of a bundle. [It appears basic DTN2 does not use this event. Not needed for N4C.] • Post a *bundle attributes report* event ensuring that the referenced bundle will not be expired before the posted event completes (see item 71).

71.	bundle attributes report	Event used to return the attributes of a specified bundle to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the bundle attributes information returned.] • No action required by Bundle Daemon Core. • Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.
72.	registration added	This event is generated when the API or an internal application adds a new registration. • Scan all bundles in the *pending bundles* list and queue any that are not fragments with a matching destination for delivery to the appropriate API client (see item 117).
73.	registration removed	This event is generated when the API or the command interface removes a registration. • Delete the *registration* from the *registration table* (see item 19). • Post a *registration delete* event (see item 75).
74.	registration expired	This event is generated when the lifetime of a *registration* expires. • Marks the registration as expired. • If the registration is not *active* (i.e., bound to an application), immediately delete the registration from the *registration table* (see item 19) and post a *registration delete* event (see item 75). • If the *registration* is *active*, it will be removed when it goes *passive* or is actively removed by the application/user using it.
75.	registration delete	Follow on event generated after a *registration* has been expired or removed (see items 73 and 74). Assumes the registration is not active. • Unconditionally delete the registration object. If there were any bundles queued for deferred delivery on this registration, they will all be dequeued and left to expire.

76. contact up

Event generated by a convergence layer indicating a contact is now open for communication (see item 272).

- Set the state of link on which the contact is made to OPEN (see item 252) - unless there has been some sort of race condition as a result of a stale notification.
- The *Contact Manager* processes this event (see item 349).
- The current *Router* processes this event (see item 192).

77. contact down

Event generated by the contact manager indicating a contact is no longer available. for communication

- Update the link uptime statistics.
- The current *Router* processes this event (see item 193).

78. contact query

Event generated by External Routers to request a list of current *contacts*. [It appears DTN2 does not use this event apart from forwarding from the External Router interface. Not needed for N4C.]

- Post a *contact report* event (see item 79).

79. contact report

Event used to return a list of current *contacts* to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the contact list returned.]

- No action required by Bundle Daemon Core.
- Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.

80. link created

Event generated from the *contact manager* to indicate a new link has been established. Most work is done in the router for this event.

- Check that the link hasn't already been deleted - if it by any chance has, the *router* and *contact manager* handlers are bypassed.
- The *Contact Manager* processes this event (see item 346).
- The current *Router* processes this event (see items 195 and 217).

81.	link deleted	Event generated from contact manager to indicate a link has been destroyed. • Nothing to do in Bundle Daemon Core. • The *Contact Manager* processes this event (see item 343). • The current *Router* processes this event (see item 196).
82.	link available	Event generated from contact manager to indicate a link has become available. • Check that the link hasn't already been deleted - if it by any chance has, the *router* and *contact manager* handlers are bypassed. • The *Contact Manager* processes this event (see item 347). • The current *Router* processes this event (see item 194).
83.	link unavailable	Event generated from contact manager to indicate a link has become unavailable. • Nothing to do in Bundle Daemon Core. • The *Contact Manager* processes this event (see item 348).

84.	link state change request	Event generated by the Control Interface, the Contact Manager, various convergence layers, the Discovery components and the External Router to notify link state changes to the rest of the BD. The possible states for links are listed and described in Section 4.9.1.4). • Check the link is still present (there can be race conditions with deletion) and not stale. • Perform sanity checks so that the link state transitions are sensible according to the link state machine. The allowed transitions are documented at item 252. Then call action routines accordingly. The main complexity is when the link is closing down. • If transitioning to UNAVAILABLE state, post a *Link Unavailable* event (item 83) for the bundle core. • If transitioning to AVAILABLE state, post a *Link Available* event (item 82) for the bundle core at the head of the queue so it is processed next. • If transitioning to OPEN state or OPENING state, call the *open a link* functionality (see item 111). • If transitioning to CLOSED state, call the *close a link* functionality (see item 112). [In DTN2 this functionality is mostly embedded in the event handler.] In practice the CLOSED state is transient. After executing the *close a link* function, the link will be in AVAILABLE or UNAVAILABLE state.
85.	link create	Event generated by External Router to request creation of a new link. • Check no link with this name exists • Extract parameters from request • Create new link object • Inform contact manager of new link (see item 342).

86. link delete request

Event generated by External Router, External Convergence Layer and Command Interface to request deletion of an existing link. The External Convergence Layer has to do this deletion if the convergence layer itself is destroyed, when any links created through discovery using the convergence layer have to be deleted.

- Assuming the link is still in existence, call the contact manager to delete the link and associated information (links maintain information used by both the associated convergence layer and the router and which is understood by the other component but is specific to the link - see item 343).

87. link reconfigure

Event generated by External Router to request reconfiguration of the parameters of an existing link.

- Send new parameters to Link reconfiguration routine(see item 250).

88. link query

Event generated by External Routers to request a list of current extant links. [It appears DTN2 does not use this event apart from forwarding from the External Router interface. Not needed for N4C.]

- Post a *link report* event on the Bundle Daemon Core (see item 89).

89. link report

Event used to return a list of current extant links to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the current open links list returned.]

- No action required by Bundle Daemon Core.
- Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.

90. link attributes query

Event generated by External Routers to request the attributes of a specified link. [It appears DTN2 does not use this event apart from forwarding from the External Router interface. Not needed for N4C.]

- Post a *link attributes report* event on the Bundle Daemon Core (see item 91).

91. link attributes report

Event used to return the attributes of a specified link to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the link attributes information returned.]

- No action required by Bundle Daemon Core.
- Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.

92. reassembly completed

Event generated by the fragment manager (see item 171) to signal that a bundle just received was the last piece needed to complete reassembly of a whole bundle. Both this event and the resulting *bundle received* event are posted at the head of the queue so that the effect is continue immediate processing of the whole bundle for which the fragment is the last 'piece of the jigsaw'.

- Delete all the received fragments that were reassembled into this bundle.
- Post a *bundle received* event at the head of the queue for the reassembled bundle, quoting the source as EVENTSRC_FRAGMENTATION (see item 56).

93. route add

Event generated by Command Interface to request addition of a new manually configured route.

- The Bundle Daemon Core does not have to do anything as this event is handled by the current router. Note that not all routers handle manually configured routes. [At present some DTN2 routers (e.g., PRoPHET and DTLSR) will silently ignore route addition requests.]
- The current *Router* processes this event (see item 190).

94. route delete

Event generated by Command Interface to request deletion of a manually configured route.

- The Bundle Daemon Core does not have to do anything as this event is handled by the current router. Note that not all routers handle manually configured routes. [At present some DTN2 routers (e.g., PRoPHET and DTLSR) will silently ignore route deletion requests.]
- The current *Router* processes this event (see item 191).

95. route query

Event generated by External Routers to request the a list of all active routes in the current *route table*. [It appears DTN2 does not use this event apart from forwarding from the External Router interface. Not needed for N4C.]

- Post a *Route Report* event (see item 96).

96. route report

Event used to return a list of all active routes in the current *route table* to the External Router. [It appears basic DTN2 does not use this event. Not needed for N4C. The External Router interface handles the generation of the routes list returned.]

- No action required by Bundle Daemon Core
- Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.

97. custody signal

Event generated by the internal Administration Application (see Section 4.12.5) when a administrative bundle of custody signal type is received and delivered to the application.

- Determine if the report applies to a bundle for which this node has custody.
- If the delivery of the bundle to a new custodian was successful or the original bundle was delivered redundantly, release custody (see item 126) and try to delete the bundle (see item 118).

  (The redundant delivery case is somewhat strange - it means that the bundle for which we have custody arrived at its new custodian twice, but the custody report for the original bundle has probably got lost. If the original custody report arrives later it will probably be flagged as an error because the bundle to which it applied will have disappeared.)

98.	custody timeout	Event generated when the *custody timer* for a bundle for which this node has custody expires. • Since the *custody timer* associated with a (bundle, link) pair has expired, delete the timer. • Modify the forwarding log to replace the *transmitted* indication with a *custody timeout* indication. This will allow the router to transmit the bundle again on the same link. • The event is processed by the current *router* which should update the forwarding log and, depending on available resources, schedule the bundle for retransmission (see item 197). [Note that in DTN2 not all routers appear to handle the custody timeout event. PRoPHET, TCA and DTLSR are examples.]
99.	shutdown request	Event generated by DTN server to request the Bundle Daemon Core thread to shut itself down (see Sections 4.5.1, 4.5.2, and 4.5.3). • Close down any open links. • Call shutdown procedures registered for convergence layers, router and the application as a whole. • Flag the run loop to exit.
100.	status request	Event generated by the Command Interface to request the status of the BD (see item 371). • This event should probably return some sort of summary of the current dynamic configuration. Just returning as DTN2 does at present indicates that the BD main thread event loop is actually running.
101.	cla set params	Event generated by the External Router to set the parameters of a specific convergence layer. [Not used by N4C.] • Pass the parameters to the convergence layer
102.	bundle queued query	[This event does not currently seem to be used in DTN2. It looks as if this is a piece of almost orphaned functionality discarded during the development of the External Convergence Layer component.] • Checks if a specified bundle is queued on a given link and posts a *Bundle Queued Report* event to give the answer (see item 103).

103. bundle queued report

Event generated by *bundle queued query handler*. [The generating event is not currently used in DTN2.]

- Does nothing apart from log result of query.

104. eid reachable query

[This event does not currently seem to be used in DTN2. It looks as if this is a piece of almost orphaned functionality discarded during the development of the External Convergence Layer component.]

- Queries the convergence layer associated with an interface to see if a given eid is currently reachable.
- The convergence layer posts an *eid reachable report* event after determining reachability [except that the External Convergence layer only posts this event on an error path through the code. On the 'good' path it posts a different internal convergence layer event that is local to the External Convergence layer .]

105. eid reachable report

Event generated by eid reachable query handler. [The generating event is not currently used in DTN2.]

- Does nothing apart from log result of query.

106. link attribute changed

Event generated by External Convergence Layer to request updating attributes of a specified link. The event is also processed by the External Router.

- Check link is still present
- If so, request update of one or more link attributes (currently *nexthop*, *how_reliable* and *how_available*).
- Will be processed by current *router* if it is the External Router which passes the message back across the XML-encoded interface.

107. iface attributes query

[This event does not currently seem to be used in DTN2. It looks as if this is a piece of almost orphaned functionality discarded during the development of the External Convergence Layer component.]

- Query the attributes of a specified interface through its associated convergence layer.
- The queried convergence layer posts an *iface attributes report* event (see item 108) [except that the External Convergence layer only posts this event on an error path through the code. On the 'good' path it posts a different internal convergence layer event that is local to the External Convergence layer.]

108. iface attributes report

Event generated by convergence layers in response to an *iface attributes query* message. [The generating event is not currently used in DTN2.]

- Just log what query was made.

109. cla parameters query

[This event does not currently seem to be used in DTN2. It looks as if this is a piece of almost orphaned functionality discarded during the development of the External Convergence Layer component.]

- Query the parameters of a given convergence layer.

- The queried convergence layer posts an *cla parameters report* event (see item 110) [except that the External Convergence layer only posts this event on an error path through the code. On the 'good' path it posts a different internal convergence layer event that is local to the External Convergence layer.]

110. cla parameters report

Event generated by the routine that implements the *cla parameters query* event handler in convergence layers. [The generating event is not currently used in DTN2.]

- Just log what query was made.

4.5.5 Additional Functionality Supporting the Event Handler

This section details a number of functions that are invoked from the event handlers to perform specific more complex functions in response to events. This functionality may be called from more than one event handler in some cases. The functions are:

- Item 111: Open a link.
- Item 112: Close a link.

- Item 113: Add a bundle to the *pending bundles* queue.
- Item 114: Queue a bundle for transmission on a link.
- Item 115: Attempt to cancel a bundle previously queued for transmission on a link.
- Item 116: Check if a bundle is to be delivered locally.
- Item 117: Deliver a bundle to a local registration.
- Item 118: Try to delete bundle.
- Item 119: Delete a bundle.
- Item 120: Delete a bundle from the *pending bundles* queue.
- Item 121: Generate a *status report* relating to a bundle.
- Item 122: Parse incoming *status report*.
- Item 123: Generate a *custody signal* for a bundle.
- Item 124: Parse incoming *custody signal*.
- Item 125: Accept custody of a bundle.
- Item 126: Release custody of a bundle.

111. **Open a link**. Check that the link is not in process of being deleted. Check that that the link is not already open or has an associated contact already. Check that the link is not in the UNAVAILABLE state. If all these checks succeed call the link *open* functionality (see item 253). Otherwise log the event and do nothing. The result of calling the *open* function depends on the type of link involved. For non-connection type links this will typically result in the link immediately becoming open because the functionality is all local, but for connection oriented link types, network operations are involved with round trip delays. In this case the link is placed into the OPENING state pending completion of session initiation.

112. **Close a link**. A reason code provided by the Contact Manager is passed in as a parameter. Check that the link is currently OPEN or in the process of OPENING. If not, log the event and do nothing more. If the check succeeds, and the link is OPEN (rather than OPENING) post a *Contact Down* event for the bundle core (see item 77) to indicate that the contact is no longer active. Then call the link *close* functionality (see item 254). Depending on the reason code supplied transition the state either to AVAILABLE state if the link has closed because it has been idle for too long (*on demand* type links only) and otherwise to UNAVAILABLE state. If the state is now UNAVAILABLE, post a *Link Unavailable* event on the bundle core for immediate processing (see item 83).

113. **Add a bundle to the *pending bundles* queue and, if requested, store the bundle in persistent storage**. Push the bundle supplied onto the tail of the *pending bundles* queue (see item 32). If requested store the bundle in persistent storage (storing in persistent storage clearly has to be skipped when bundles are being read back in from persistent storage at start-up; it will also be skipped for very few transient bundles that are created internally and sent out immediately. [DTN2 only needs this capability for the external router and possibly the DTLSR router].) From the information in the bundle, determine the *expiration time* of the bundle (absolute creation time plus the lifetime period (expiration)). If this time is already in the past as determined by the local system clock (take care to use UTC) then the bundle has already expired, so set the expiration time as *now*. Schedule the expiration timer for the bundle to expire at the expiration time. New bundles on the *pending bundles* queue that have not yet expired should be notified to the router so that it can

determine what to do with them. Return a flag to indicate if the bundle had expired so that the event handler can decide whether to inform the router.

114. **Queue a bundle for transmission on a link**. If this BD has custody of the bundle, a *custody timer* will be passed in to allow a later retransmission to be scheduled if the BD at a subsequent hop has not reported taking custody of the bundle before the timer expires. Check that the link is not in process of being deleted. If it is being deleted log a message and do not queue the bundle. Otherwise check that the wire format has not already been constructed for this link. This indicates that the bundle is already in process of transmission on this link If it is already in process, abort the queuing. Otherwise create the wire format of the bundle for the specified link. The format may depend on the security, maximum bundle size (fragmentation threshold) and other policies associated with the link. The creation of the blocks is a two stage process (preparation of the list of required blocks and generation of the wire formats for the resulting list - see item 150). Check that the latest entry in the *forwarding log* for the bundle does not show that it was lately queued on the link. Check that the generated wire format is not too large for the MTU (Maximum Transmission Unit) of the link. Check that the bundle is not already in either the *sending queue* or *inflight* queue for the link. If any of the checks fails abort the queuing. [DTN2 note: Should the wire format for the link be deleted again? Strikes me we might get either a deadly embrace if we try to send the bundle again later because the link blocks are already in place or at least a memory leak because nothing clears up.] Otherwise add an entry to the *forwarding log* to show the bundle has been queued on the link and recording the custody timer. Then add the bundle to the link's *send queue* (see item 257) and notify the convergence layer used on the link that a new bundle has been queued (see item 274).

115. **Attempt to cancel a bundle already queued for transmission on a link**. Check that the link is not being deleted. Check that a set of wire format blocks have been created for the link, indicating that it is either on the *send queue* or *inflight*. Do nothing if either check fails. Otherwise, check if the bundle is still on the *send queue* of the link. If so attempt to delete the bundle from the *send queue* (see item 258). If this is successful, post a *Bundle Send Cancelled* event with the bundle daemon (see item 63) and return. Otherwise check if the bundle is on the *inflight queue*. Depending on the type of convergence layer and how far along transmission has progressed some convergence layers may be able to cancel sending. There is no guarantee that this is possible. Request the convergence layer to attempt the cancellation (see item 275). Otherwise log a warning as the bundle is not on either queue which it should be.

116. **Check if a bundle is to be delivered locally, and optionally carry out any deliveries**.. Note that only complete bundles can be delivered - fragments should not arrive here. Scan the list of *registrations* to see if there are any that match the destination EID of the bundle (see item 20). If there are any and delivery is requested, perform the delivery (see item 117). Return true if any *registrations* matched or if the destination EID is 'subsumed' by the local EID, i.e., the destination is potentially on the local node whether or not there is a registration for the service tag in the destination EID. This check is done to identify bundles that possibly should not be passed on to the router for forwarding.

117. **Deliver a bundle to a local *registration***. Check that the most recently logged forwarding information for the bundle does not indicate that it has already been delivered. If this bundle is a duplicate of a bundle that has already been delivered as indicated by the *registration's delivery cache*, suppress delivery. Otherwise deliver the bundle to the registration (see item 20) and log the delivery in the bundle's *forwarding log*.

118. **Try to delete a bundle**. This functionality is provided to allow *early deletion* of bundles that are apparently no longer required but would otherwise hang around the system until they hit expiry time. The BD can be configured to allow early deletion of bundles (see item 34) if the router in use consents. If the bundle has not already expired, the BD is configured to allow early deletion and the router says it is permitted (see item 205), delete the bundle before its expiry time (see item 119).

119. **Delete a bundle**. A reason code may be provided. Add the deletion to the BD statistics. If this BD has custody of the bundle or if a deletion status report was requested in the bundle flags and a reason for deletion was provided then a status report should be sent if the bundle is actually deleted. If we have custody of the bundle, release it (see item 126). If custody transfer had been requested but the BD was unable to comply because of resource constraints or a validation error, send a custody failed report. [DTN2 does not do this.] If the bundle is a fragment delete any existing fragmentation state relating to the bundle (see item 170). Notify the router(s) that the bundle is being deleted (see item 206). Delete the bundle from the *pending queue* (if present on the queue). If deletion was successful and a status report is required, send the *deletion status report* (see item 121).

120. **Delete a bundle from the *pending bundles* queue**. Thus is the final stage prior to deleting a bundle completely. If the bundle still has an *expiration timer*, cancel it and delete the timer. Delete the bundle from the *pending bundles* queue. Once this is happened the bundle would normally expect to be only referenced in the *all bundles* list and will consequently be ready to be deleted from persistent storage and the memory version disposed of (see Section 4.6.2).

121. **Generate a *status report* relating to a bundle**. Encode an *administrative bundle* that has a *status report* as its *payload*. The format of the payload is defined in Section 6.1.1 of [RFC5050]. The bundle header fields are set as follows:
    - Source EID is generally the local EID of the reporting BD.
    - Destination EID is the *reply to* EID in the bundle being reported on if it is not the null EID (dtn:none) or else the source Eid of the bundle being reported on.
    - The reply to and custodian addresses are set to the null EID.
    - It is flagged as an administrative bundle.
    - The expiration time interval is taken from the bundle being reported on.

    The bundle is then treated as if it had been just received , calling the *Bundle Received* event code. (see item 56) with the event source being set to EVENTSRC_ADMIN.

122. **Parse an incoming *status report***. Check the payload of an incoming bundle flagged as an *administrative bundle* containing a *status report* as payload matches the

specification in Section 6.1.1 of [RFC5050] and build a data structure containing the returned information. Return true (matches) or false, and the extracted data.

123. **Generate a *custody signal* for a bundle**. If we don't already claim custody of the bundle but the *custodian EID* for the bundle is not the null EID, generate a *custody signal* (an administrative bundle sent from the unqualified local EID (i.e., with no *service tag* - the *administrative EID* of the node)). The payload of the custody signal is defined in Section 6.1.2 of [RFC5050]. The bundle header fields are set as follows:
    - Source EID is the unqualified local EID (administrative address) of the reporting BD.
    - Destination EID is the *custodian* EID in the bundle being reported that may not be the null EID (dtn:none).
    - The reply to and custodian addresses are set to the null EID.
    - It is flagged as an administrative bundle.
    - The expiration time interval is taken from the bundle being reported on.

    The bundle is then treated as if it had been just received , calling the Bundle Received event code (see item 56) with the event source being set to EVENTSRC_ADMIN.

124. **Parse an incoming *custody signal***. Check the payload of an incoming bundle sent to the *administrative EID* of the node matches the specification in Section 6.1.2 of [RFC5050]. Return true (matches) or false otherwise.

125. **Accept custody of a bundle**. Check that the BD does not already have custody and that the *custodian EID* is not the local EID. If not, and if the bundle *custodian EID* is not the null EID, generate and send a *custody signal* to the custodian EID (see item 123), then change the custodian address to be the local EID, mark the bundle as being in local custody, update the persistent storage to reflect the changes, add the bundle to the list of *custody bundles* (see item 32) and if the originator of bundle requested custody acknowledgements generate and send an appropriate *status report* (see item 121).

126. **Release custody of a bundle**. Check that the BD has local custody of the bundle. If so cancel any active custody timers associated with the bundle, set the custodian address to the null EID , clear the local custody flag and update persistent storage to reflect the changes. Remove the bundle from the list of bundles for which the BD has custody (see item 32).

4.6 BUNDLE FACTORY AND FRAGMENTATION MANAGER

The components in this section deal with DTN bundles as defined in the Bundle Protocol [RFC5050] and the Bundle Security Protocol [BSP] together with various separately defined extension blocks. Bundles can either be complete bundles or bundle fragments. Both complete bundles and fragments are eligible for storage and forwarding. Only complete bundles can be delivered to local applications and the *Fragmentation Manager* is responsible for organizing sets of fragments of a bundle and determining when the available fragments 'cover' a bundle so that it can be reassembled and delivered.

The fragmentation manager can also split up a bundle into fragments either proactively because the bundle exceeds a configured limit on bundle size for a specific link *proactive fragmentation)* or reactively if a link goes down in the middle of bundle transmission - the fragments may be created on both the transmitting side and the receiving side of the link. *Reactive fragmentation* is only possible if the link in use features acknowledged, reliable transmission so both sides are aware of what has happened and can split a bundle without losing data at the split. A bundle can only be fragmented at a point in its payload, and each fragment must contain at least one octet of the payload, meaning that bundles with no payload or a payload length of one cannot be fragmented.

The *Bundle Factory* creates a data structure that represents the bundle in an internal form that makes it convenient to manipulate the individual fields of the bundle but has to be converted to and from the various other formats in which a bundle can be represented. In particular the bundle has to be converted to and from the format in which it is passed 'over the wire' when being transmitted over a network.

Overall the fragmentation manager looks after incoming and outgoing bundles dealing with necessary fragmentation and reassembly. The fragmentation manager feeds completed received bundles to bundle core for delivery or forwarding.

When the *bundle security protocol* is in use, *bundle reassembly* may also be necessary at some intermediate points in order to verify the integrity of a bundle if the bundle was split at a link that was using hop-by-hop integrity checking.

127. The BD must ensure that bundles are stored in persistent storage unless specifically requested otherwise by the creator of a bundle.

128. Bundles and associated state must be saved and restored across restarts of the BD, and so far as is possible, should be preserved in the event of a uncontrolled termination of the BD.

4.6.1 Bundle Structure

DTN bundles are a sequence of blocks of various different types. Each bundle starts with a primary bundle block, which must be the only primary block in the bundle, and must contain at least one other block. The last block in the bundle contains a flag that identifies it as the last block.

The following block types are defined and need to be handled by the BD:

- Primary Block [RFC5050]
- Payload Block [RFC5050]
- Extension blocks as per RFC 5050:
  - Metadata Block [MetadataBlock]
  - Previous Hop Insertion Block [PrevHopBlock]
  - [Session Block - supported experimentally in DTN2 but not required for N4C and not standardized.]
- Security related blocks as per Bundle Security Protocol [BSP]:
  - Bundle Authentication Block (BAB)
  - Payload Integrity Block (PIB)
  - Payload Confidentiality Block (PCB)
  - Extension Security Block (ESB)

[Note that version 2.7.0 of DTN2 implements an earlier version of the Bundle Security Protocol than is currently being standardized. The names of the PIB and the PCB were respectively Payload Security Block (PSB) and Confidentiality Block (CB) in this version (approximately version -04 of the draft). Also the ESB had not yet been introduced and is not implemented in DTN2 at this time. This affects the configuration interface described in item 482.]

For details of the structure of each block refer to the relevant RFC or draft.

There are a number of additional block types and mechanisms being developed in the DTN Research Group, but these are either still at an early stage of development or are considered inappropriate for the N4C environment. These include the Retransmission Block [RetransBlock] (not clearly useful in N4C), Compressed Bundle Header Encoding [CBHE] (essentially specific to the space community and of limited application to the N4C environment unless it finds application in the N4C telemetry applications), the Extended Class of Service extension block [ECOS] (consideration will be given to incorporating this mechanism later as a way of providing a broadcast in a DTN), and Reliability-only Ciphersuites [Checksum] (suggested adjunct to custody transfer; not yet accepted as a useful solution).

129. **The BD must be able to receive, forward and deliver bundles that contain blocks of types that it may not be able to explicitly process**. All blocks have a *canonical block header* defined in Section 4.5.2 of [RFC5050] that allows their type to be identified, the block flags to be examined and the length of the block to be known. The body of the block can be treated as opaque data for this purpose. The block may contain EID references to names in the *Primary Block EID dictionary*. These must be correctly treated during conversion between internal and wire format representations. Bundles containing blocks of unknown types should be forwarded with the unknown blocks unchanged unless the *Discard Bundle if Unable to Process* flag is set in the bundle flags or *Discard Block if Unable to Process* flag is set in the block, but with the *Block Forwarded Unprocessed* flag set in the block. A node that does not support the Bundle Security Protocol (BSP) will need to treat blocks of the types introduced in the BSP as unknown types. In particular, *Bundle Authentication Blocks* must not be forwarded as they are sent hop-by-hop and not end-to-end.

[DTN2 does not discard bundles if the *Discard Bundle* flag rather than the *Discard Block* flag is set.]

4.6.2 Bundle Construction and Dissection

Any bundle and its component blocks can be represented in four different forms:

- Internal object – the master format,
- Network wire format as defined in RFCs and Internet Drafts,
- API specification, and
- Persistent storage serialization.

This component provides methods to manage the conversion of bundles between formats and maintain the master object structures.

Except in some very limited circumstances, all bundles will be backed up in persistent storage as soon as they are created, and if they are subsequently modified. At present the persistent storage associated with a bundle only needs to be altered when custody is accepted or released on a bundle. The persistent storage is deleted when the bundle is deleted.

[In DTN2 bundles are reference counted objects. This means that they can be 'garbage collected' and the persistent storage released when the last reference to a bundle is removed. To assist with this, all bundles are added to the *all bundles* list the first time a reference is created. When the number of references drops to one because it is just in *all bundles* the reference deletion code posts a *Bundle Free* event to the bundle core which cleans up the storage. Sometimes I think C++ is much too clever for its own good.]

The following functions are provided to manage bundles overall:

130. **Create a new bundle**. Assign a new bundle ID for the bundle. The bundle ID is a sequence number that is unique across all bundles that have ever been created for the associated persistent data store. The latest value of the sequence ID is stored in the persistent datastore and used to continue the sequence if the BD is restarted. Fills in default values for all fields.

131. **Reconstitute a bundle from stored data in the persistent data store**. The bundle ID retrieved from persistent storage will be assigned to the bundle (see item 154).

4.6.2.1 Internal Representation

The internal representation of a bundle maintains an image of the information that makes up the primary block as described in [RFC5050], information about the payload of the bundle [In DTN2 the payload is stored in a separate platform operating system file. The Internal Representation records the file name of this file] and two vectors of block structures describing the other blocks in the bundle. The blocks are separated into blocks received from the wire and those received from the API. This allows them to be treated differently by the security protocol. Blocks received from the wire will (in many cases) not need to be reprocessed by the security suites before onward forwarding - just as well because intermediate nodes won't have the relevant keys anyway! Blocks coming from the API will be *en clair* and will need to be processed by the security suite before being

forwarded. The internal representation also holds information about the bundle state within the BD. The information held is

- Local bundle identifier.
- Indicator of whether the bundle is up to date in persistent storage.
- Indicator of whether the local node has custody of this bundle depending on the bundle flags.
- The forwarding log, recording the history of this bundle as it is received and forwarded on various links. [Note that this log should be persistent, but DTN2 does not currently maintain this log across restarts. This means that the restarted router does not know whether bundles should be resent when reloaded from persistent store, and custody timers are lost. The reason for this is that the DTN2 does not have persistent names for links.]
- Expiration timer for the bundle. If the timer expires, the bundle will be deleted unless the router has specifically requested otherwise (section 4.5.4, Item 59).
- Vector of custody timers for the bundle. There is a custody timer per link on which a bundle has been transmitted, created when the bundle is recorded as having been transmitted. If a custody signal indicating successful custody transfer is received (Section 4.12.5 and item 97) all the custody timers are cancelled and the bundle is deleted. If a custody timer expires generating a *custody timeout* event (see item 98), the forwarding log is updated to record the timeout and the router is expected to schedule a retransmission if policy requires this, depending on the number of retransmissions allowed. See Section 4.6.5 for information about how custody timers are configured. [Some thought is required here to decide how custody timers should be managed for opportunistic contacts. A link is associated with the next-hop in opportunistic network situations. If the custody timer expires, should one wait for the same contact to come round again or treat the custody timer as requiring retransmission on *any* link that becomes available? This is a router issue, but affects the identification of the timer/link pair when the timer expires here. Should the failure affect delivery predictabilities in PRoPHET?]
- Set of wire format transmission block vectors per transmission link with a transmission in progress. The reason that there is not just one wire format vector is that some metadata blocks may be flagged to be not transmitted on particular links and have to be left out of the data to be sent. [DTN2 has means to record that metadata blocks can be added to a bundle anywhere along its forwarding path(s) and to allow such metadata blocks to be only sent on certain links. However there is no API to control this and the metadata would currently be sent end-to-end. There is also some doubt that the metadata block processor matches the most recent draft [MetadataBlock] as regards data length specification.]

4.6.2.2 Bundle Structure Verification

The various specifications place a number of constraints on the structure of bundles, and the flags that can be set in the various blocks that make up a bundle. In the course of construction of bundles whether from API instructions, data received from the network or by retrieval from persistent storage, all bundles must satisfy the following constraints:

132. Bundles are constructed from a single *Primary Block* that must be the first block in the bundle and a number of other blocks as specified in Section 4 of [RFC5050]. A bundle must contain a *Primary Block* and at least one other block.

133. The *Primary Block* structure must be as defined in [RFC5050]. It is completely specified in RFC 5050 Sections 4.2, 4.5 and 4.5.1.

134. The *EID dictionary* in the *Primary Block* must consist of concatenated null terminated strings as specified in Section 4.4 of RFC 5050. Each string must be no longer than 1023 octets. Any non-zero octet may be used within the string. References to the entries in the EID dictionary specify a pair of offsets into the dictionary denoting the *Scheme* part and *Scheme Specific* part of the EID. Each offset must reference a position immediately after a null (zero) octet on the dictionary.

135. All other blocks must contain the standard preamble defined in [RFC5050], Section 4.5.2 (the *Canonical Block Header*). This starts with an 8 bit unsigned integer that defines the block type and an *SDNV* (Self-Delimiting Numeric Value) containing the block processing control flags for the block defined in RFC 5050, Section 4.3. This may optionally be followed by one or more EID references pointing to the *EID dictionary* in the *Primary Block*. If any such references are present, a flag is set in the block processing control flags. The preamble is completed by the SDNV containing the block data length that specifies the number of octets in the data field that follows the preamble.

136. Bundles contain either zero or one *Payload Block*s. The *Payload Block* is specified in [RFC5050], Section 4.5.3. Note that *Payload Blocks* must not contain any EID references.

137. Any other blocks are generically known as *Extension Blocks* and the data field may be structured according to the block type. Particular validation may need to be applied to blocks of this type according to their definitions.

138. If the *bundle processing control flags* indicate that the bundle's protocol data unit (the body of the *Payload Block*) is an *administrative record*, then the *custody transfer requested* flag must be clear and all status report request flags must be clear.

139. In the current version of the BD, in line with RFC 5050, the *bundle processing flags* should be a 3 octet SDNV (21 flag bits) with bit positions 6, 9 to 13 and 19 onwards reserved for future use and thus normally be cleared. Also having both bits 7 and 8 set is not a valid bundle priority value.

140. If the bundle's *source endpoint ID* is ‘dtn:none’, then the bundle is not uniquely identifiable and all bundle protocol features that rely on bundle identity must therefore be disabled: the bundle's custody transfer requested flag must be zero, the *Bundle must not be fragmented* flag must be set, and all status report request flags must be cleared.

141. If the processing flags in the *Primary Block* indicate that the bundle payload is an administrative record, then the *Transmit status report if block can’t be processed* flag

of every other block in the bundle must not be set to avoid potential error cascades, as specified in [RFC5050] Section 4.3.

142. The *Block must be replicated in every fragment* flag must not be set in any block that follows the Payload Block. This requirement is made to avoid a Catch 22 situation associated with *reactive fragmentation*: only the bundle payload can be split across fragments; if a partially received bundle were reactively made into a fragment it would be impossible to tell if any of the subsequent blocks after the partially received payload had the 'must be replicated' flag set, making it impossible to consistently create a fragment.

143. In the current version of the BD, in line with [RFC5050], the *block processing flags* should be a one octet SDNV. All bits (0-6) are currently used; bits 7 onwards are reserved for future use and thus normally cleared.

144. For Metadata Blocks [MetadataBlock], only the *URI Metadata* type is supported. If a Metadata Block is included in the bundle, it is expected to contain a single *URI Metadata* component in the data field. The block must not contain any EID references. The data field should be made up of a concatenation of null terminated strings: the last data octet must therefore be a null. In future new *Metadata types* may be defined. Depending on the setting of the bundle and block processing flags, the BD can either ignore and forward unprocessed Metadata Blocks that it does not understand or delete either the block or the whole bundle.

145. For Previous Hop Insertion Blocks [PrevHopBlock] the block must have the *Discard block if it can't be processed* flag set. Setting this flag prevents the block being forwarded for more than one hop even if the receiving node does not understand Previous Hop Insertion Blocks. The *Block was forwarded without being processed* flag must not be set. The block must either

    145.1 contain exactly one EID reference in the preamble identifying the previous hop EID in the *Primary Block EID dictionary*, with the *Block contains an EID reference field* flag set, and the *Block Data Length* must be zero (i.e., the block does not contain the EID); or

    145.2 contain zero EID references in the preamble with the *Block contains an EID reference field* flag clear, and the *Block Data Length* must be non-zero. The Block data field must contain two null terminated strings which must not be in the Primary Block dictionary. This implies that the last octet of the block must be a null octet.

146. The bundle structure appropriate to any security cipher suite(s) in use must be correct as specified by the BSP. See Section 4.6.3.

4.6.2.3 Wire Format Conversion

Bundles will be received and sent via the *convergence layer* selected for the *link*. In many cases the *link* will send data in chunks such as IP packets or Ethernet frames that will be smaller than most bundles. Thus the wire format conversion needs to be able to consume and generate the on-the-wire representation in chunks dictated by the *convergence layer*.

Also, bundles may be very large and in some memory resource constrained environments it may be inconvenient or impossible to keep multiple different complete representations of a bundle in memory at a time (e.g., large image files as used for returning data from spacecraft may be hundreds of megabytes or even larger). Performing the conversion in chunks can obviate this problem. Note that when encrypting a bundle, it will often be convenient to use a streaming cipher to avoid needing to handle the whole data structure in memory.

147. **Provide a function to consume an arbitrarily sized chunk of data and either start a new bundle or append it to an existing bundle block under construction.** All blocks contain a standard preamble that specifies the type of the block in the first octet and the length of the block. Once the preamble has been decoded, a block object of the correct type to store the block can be created and the length of the block is known. The remainder of the block can be copied from the supplied data. The block object is linked to the bundle under construction. At this stage the block contents can be treated as opaque data. The chunk may contain more data than is required to complete the block. Any remaining data must be left intact as it should be the start of another block. This requires two passes over the data. One to build the basic blocks and a final one to validate them and complete the separation of the *Primary Block* from other blocks. The function returns the amount of data consumed and a flag indicating if the block is complete

148. **Provide a function to dissect a complete received data block into fields known by the protocol according to the various RFCs/drafts**. Blocks other than the *Primary Block* may require information (specifically EIDs) from the *EID dictionary* in the *Primary Block*. Thus the *Primary Block* has to be dissected first before any others can be processed.

149. **Provide a function to validate the contents of the block**. The *Primary Block* of a bundle must have been dissected before it is ge5nerally possible to dissect others as they need information from the *EID dictionary*.

150. **Provide functions to assemble the wire format of a bundle and store it ready for transmission**. The set of blocks to be assembled may depend on the link for which the data is being assembled. In particular one or more *Metadata Blocks* may be injected into a bundle at any node on the forwarding path. [The need for and exact use of this capability is not clear at the moment but it is specified in [MetadataBlock]. DTN2 has low level capabilities for handling this but no API to allow it to be used. It is potentially of use in connection with passing routing and forwarding information piggy backed onto bundles in an opportunistic environment.] Similarly a *Previous Hop* block can be added to a bundle at each hop across which it is sent [PrevHopBlock]. Building the wire format requires three passes over the list if blocks: to prepare the set of blocks, generate the basic data and then finalize it, possibly including encryption, as:

    150.1 The contents of the *Primary Block EID dictionary* may depend on items in other blocks.

150.2 If the *Bundle Security Protocol* is in use, encryption may alter the length of blocks, and/or result in the insertion of extra blocks in the list of blocks to hold security results. The final pass needs to be in reverse order to cope with encryption which may alter the lengths of blocks. The ultimate length has to be incorporated into the Primary Block which is transmitted first.

151. **Provide functions to access the resulting wire format to allow it to be transmitted over the wire**. This should allow the information to be accessed in arbitrarily sized chunks as with the receiving function. [DTN2 is currently designed to pre-generate the on-the-wire format when the bundle is transferred to the Link's *send queue*. In a memory constrained environment, it would potentially be possible to create the wire format 'on the fly' as it was requested by the *Convergence Layer*. However, for bundles that are being processed by ciphersuites, this may require an extra 'dummy run' over the block set to determine sizes without writing the data, followed by a repeat of the block generation when the Convergence Layer actually requests the data. This extra elaboration is not provided by DTN2.]

152. Provide functions to delete the set of blocks containing the wire format of a bundle for a given link.

4.6.2.4 Persistent Storage Serialization and Deserialization

Most bundles have to be maintained in *persistent storage* so that they can be preserved across BD restarts. The preferred mechanism involves storing all blocks and their associated data in a suitable database, except for the *payload block* data which may be very large. The *payload block* data is kept in a separate operating system file with the name stored in the *payload block database record*. As far as possible, operations should be 'transactional' so that the internal representation and the persistent storage representation remain consistent, and the persistent representation is internally consistent. Note that this may impose specific requirements on the platform operating system's filing system if payload data is stored in operating system files. [The default choice for the DTN2 database is the Berkeley Database. An SQL based database may be a better choice in the future - for example 'sqlite'. DTN2 also offers the option of storing the information completely in operating system files if the platform cannot offer a database of any kind. This could be a useful option for very small platforms where the overhead of a database is undesirable.]

153. **Provide functions to convert a bundle into the chosen database format and write it to persistent storage as a new entry**.

154. **Provide a function to retrieve all stored bundles from the chosen database and create the internal representation of the bundle**. Carry out any operations needed to retrieve auxiliary information (e.g., the bundle forwarding log) for each bundle and notify other parts of the system that the bundle is present. In particular establish the bundle's expiry timeout (it is possible that the bundle has expired while the BD was shut down, in which case the bundle should be deleted immediately, and any necessary expiry reports sent if specified in the bundle flags). Also notify the router of the bundle's existence so that any necessary forwarding actions can be carried out as links become active.

155. **Provide functions to update or delete the persistent representation of a bundle to keep in step with the internal representation should this be modified or deleted**.

4.6.2.5 API Conversion

As described in Sections 4.4.1.1.12 and 4.4.1.1.14, the *dtn_send* and *dtn_recv* API functions are used for the interchange of bundles between the BD and applications that generate and receive bundles. The API organizes the bundle data in a different way from the internal representation, and so a conversion is required. The conversion is handled in the API server when the relevant API function is called. The API format provides the bundle data as described in the parameters of the function calls, made up of

- Primary block information including source, destination and reply-to EIDs, bundle flags and reporting options as discrete parameters;
- Payload data either as a memory block or a file name;
- Vector of extension block structures; and
- Vector of metadata block structures

The conversion routines are provided by the API component (see Section 4.4).

4.6.3 Bundle Security Protocol Support

[Note: The current version of DTN2 (2.7.0) supports a version of the Bundle Security Protocol (BSP) that is no longer current (somewhere about version -05 of the draft specification [BSP]). It is expected that the specification in version 15 of the draft will be published as an experimental RFC. It is understood that Sparta in the person of Peter Lovell is working on an updated version that will support the RFC specification.]

The BSP is a flexible but fairly complex system. It supports the use of various different cipher suites for combinations of integrity protection (authentication) and encryption. Security parameters and security results are carried in specialized blocks which are inserted into the block list both after the primary block and at the end of the list. In principle security transforms and checks can be applied and/or removed at any intermediate point along the path that a bundle takes. The BSP expresses this by defining the concept of *security regions* where particular security policy is applied. To support this, certain security blocks may contain one or both of *security-source* and *security-destination* EIDs. The *security-source* indicates where the security block was generated and the *security-destination* indicates where the security should be checked and, in the case of encryption, where the encrypted text should be transformed back into clear text. The relevant security blocks are removed at the *security-destination*.

BSP processing only applies directly to bundles that are received from the network and are being prepared for transmission to the network. It is an addition to the Wire Format conversions described in Section 4.6.2.3).

156. **The BD must be configurable at compile time to either support or not support the BSP**. The remaining items of functionality in this section (items 157 - 164) will be unavailable if the BD is not configured to support the BSP. Instead block types defined by the BSP will be treated as unknown. Depending on the setting of the block processing flags (which can be interpreted even if the block type is not recognized) the block may be forwarded or delivered as opaque data or cause either

then complete bundle or just this block to be discarded as described in item 129. In the case of *Bundle Authentication Blocks* (BAB) the bundle will be discarded on reception if the BAB cannot be processed - it should not be necessary to check this specially as BABs are marked for bundle discard if the block cannot be processed at the next hop.

157. **A *Security Policy Database* (SPD) and *Security Policy Enforcement Point* (SPEP) must be implemented** to control the application of security according to destination and available security suites. The SPD must also specify how security applies to bundle fragmentation, both proactive and reactive as discussed in [BSP] Sections 3.8 and 3.9.

158. **A Key database must be implemented** to maintain sets of keys associated with specific EIDs. The keys will be used either to authenticate/encrypt outgoing bundles that will go to or through the EID and/or to authenticate/decrypt incoming bundles that have the EID as security source. Note that the security source and destination default to the bundle source and destination respectively if not explicitly specified.

159. **A security management and configuration interface must be implemented to allow *security-source* and *security-destination EID* pairs with associated security ciphersuite selections to be inserted into and deleted from the SPD**. See Section 4.11.7.13.

160. **A management and configuration interface must be implemented to allow *(key, EID, key id)* tuples to be stored securely**. See Section 4.11.7.13.

161. **When a bundle is delivered to the BD and it contains a BAB, the SPEP is activated to determine the security suite and key to use to validate the bundle based on the *security-source* (previous hop EID) and *key id***. If the bundle validates correctly it may then be locally delivered or held for further forwarding depending on the bundle *destination EID*, and the BAB is discarded. Otherwise the bundle is discarded and a *bundle deletion* status report is sent to the report-to EID if requested by the bundle flag settings.

162. **When a bundle is delivered to the BD and this node matches the destination EID or some security destination EID is a BSP block, the SPEP is activated to determine the applicable security suite depending on the *security-source* and *security-destination*, and the applicable key based on the *security source* and *key id.*** Depending on the block type and security ciphersuite the block is validated and/or decrypted as appropriate in accordance with the [BSP]. If the security operation succeeds the processed bundle may then be delivered locally or held for further forwarding depending on the bundle destination EID. Otherwise the bundle is discarded and a *bundle deletion status report* is sent to the report-to EID if requested by the *bundle flag* settings.

163. **When a bundle is scheduled for forwarding on a given link, the SPEP will be activated to determine if any security processing should be applied to the bundle before transmission**. The SPD will be consulted to determine if there are entries containing the EID of the current node as source and as destination the EID of some or all of the next-hop nodes to be used by this bundle, that are known to be

on the expected forwarding path as determined by the routing algorithm and the final destination. Effectively the SPEP needs to be aware of the *waypoints* through which the bundle is expected to pass if a security destination other than the final destination is to be used. If a relevant pair are found, the applicable security suite should be retrieved from the SPD and the appropriate Key from the Key database, and the security processing applicable to the security suite(s) applied to the bundle to make it ready for transmission. The resulting processed bundle will be specific to the selected outgoing link, and must be associated with that link ready for transmission. If security processing fails (e.g., because the key cannot be retrieved due to an inconsistency in the configuration), the bundle should be marked for deletion and a bundle deletion report sent to the report-to EID of the bundle if requested by the bundle flag settings. Note that this processing applies to bundles received from the network and bundles originated by applications at this node, including administrative bundles. Security processing will generally require the addition of blocks to the list of blocks to be transmitted. The details are described in the BSP document.

The additional BSP processing supports the Wire Conversion mechanisms for reception described in Item 147 allowing a block to be received in chunks and assembled into a complete bundle before processing, and for three pass processing before transmission as described in Item 150. The third (finalize) stage of processing needs to work on the blocks in reverse order so that the final size of the bundle can be installed in the *Primary Block*.

164. **The [BSP] specification contains extensive rules about block ordering and validation**. A BD implementation that supports the BSP must implement all the relevant rules set out in [BSP] for the cipher suites that the BD supports.

4.6.4 Bundle Fragmenter

As described in [RFC5050] bundles may be *fragmented* either on first transmission or at some subsequent forwarding node. Fragmentation involves dividing the blocks other than the *Primary Block* into two or more sets that are communicated as separate fragment bundles. The complete bundle is reconstructed at the eventual destination when a covering set of fragments have arrived.

Bundles may be fragmented for a number of reasons. The first set of reasons utilise *proactive fragmentation* where there is a high probability, or indeed certainty, that a bundle cannot be communicated in one piece:

- If the *Maximum Transmission Unit* (MTU) of the intended transmission link is less than the bundle size (in wire format) and the convergence layer in use does not support segmentation;
- If the predicted duration of a communication opportunity combined with the expected transmission rate indicates that the whole bundle could not be transmitted during the upcoming communication opportunity;
- If the sender is informed that a bundle has been partially received and some part needs to be retransmitted; or
- If administrative policy restricts the size of bundles, for example to allow easier management of storage resources at intermediate nodes.

The second set of reasons would require *reactive fragmentation* when it is not possible to predict in advance that the whole bundle would not get transferred. At present reactive fragmentation will only be attempted in the first of these cases when it is possible for both sender and receiver to know what part of the bundle has been successfully received because they are using a reliable, acknowledged transport for the bundle transfer.

- If a communication opportunity terminates unexpectedly during transmission of a bundle (for example, this would the case when using a low latency bidirectional link when both sender and receiver are aware that the bundle has been only partially transferred and are using a reliable protocol);
- If a bundle is only partially received because of transmission path problems or receiving equipment issues (for example on high latency, unidirectional links, the sender may be unaware that the receiver has been unable to receive the whole bundle and will need to be explicitly requested to retransmit all or part of a truncated bundle); or
- If the receiver runs out of resources during reception of a bundle (for example because of pre-emption by a higher priority request) and has to discard part of a bundle.

[RFC5050] specifies that only the *payload block* can be split up across fragments. All other blocks must be carried complete in one or more of the resulting fragment bundles. Certain blocks may have the *Block must be replicated in every fragment* flag set, but these blocks must come before the *payload block* in the bundle as discussed in item 142. Otherwise *reactive fragmentation* would be impossible because the receiving node which generates the fragment reactively would generally not know about subsequent blocks that it ought to be replicating in any fragments it creates. Also, every fragment must contain a *payload block* with at least one octet of payload. Hence bundles with no payload block or a payload with only one octet of data cannot be fragmented (or refragmented if they are already fragments).

Note that because a bundle may be replicated and forwarded on two or more different paths, the bundle may be subject to different fragmentation on each path. It is therefore possible that the same data may arrive in multiple fragments. In particular the leading and trailing blocks from respectively before and after the payload can arrive in several different fragments; the fragmentation of the payload may also be different depending on the path taken.

The fragmentation manager provides the main point of control for the reception and transmission of bundles. Complete bundles can be considered as special cases of fragment bundles. Sets of fragments are tied together by the common bundle identifier which can be used as a hash key to locate fragments of the same bundle. The *Fragment Identifier* is the 3-tuple (could be also considered as a 4-tuple) consisting of:
(Creation Timestamp = (Bundle Creation Time, Bundle Creation Sequence Number),
Source EID,
Destination EID)

165. **Provide functionality to set an administrative limit for the size of bundles that can be transmitted.** Bundles larger than this *Fragmentation Threshold* have to be

fragmented. If the *Fragmentation Threshold* is set to zero, there is no administrative limit on the size of bundles. [DTN2 does not implement this functionality - see item 167 for more information.]

166. **Provide functionality to allow the contact manager to control the size of bundles that are offered for transmission as a consequence of knowledge about the expected volume of data that can be transferred during a communications opportunity**. [DTN2 does not implement this functionality - see item 167 for more information.]

167. **Provide functionality to support proactive fragmentation** by converting an existing complete bundle or fragment of a bundle into two (smaller) fragments by partitioning the payload block at a specified offset with the Primary Block data duplicated in the two bundles. The extension blocks are copied into one or both of the fragments. Blocks before the Payload Block are copied into both fragments if they have the '*Block must be replicated in every fragment*' flag set and otherwise into the fragment made from the lower part of the payload. Blocks after the Payload Block are only copied into the fragments made from the upper part of the Payload Block. Bundles must not be fragmented if they have the *Do not fragment* flag set, or if the *payload block* is missing, empty or is of length 1. [DTN2 does not appear to use this functionality although it is implemented. See item 396: in principle the *MTU* setting for a *Non-connection Oriented Convergence* layer should induce a *Fragmentation Threshold* but it is not used in this way currently.]

168. **Provide functionality to be used in case of the need for reactive fragmentation**. A fragment can only be created from a partially received bundle if the '*Do not fragment'* flag is not set and at least one octet of the Payload Block has been successfully received. Reactive fragmentation can only be used in conjunction with links that provide reliable, acknowledged transmission. [DTN2 does reactive fragmentation even on non-reliable convergence layers. This seems a little odd, but could be relevant with *LTP*.] If transmission stopped part way through the *Payload Block*, construct a fragment containing the truncated *Payload Block* and all blocks received before the Payload Block. If transmission stopped while receiving an extension block after the Payload Block, construct a fragment which contains the whole payload except for the last octet but discard the subsequent extension blocks - a fragment must contain at least one octet of payload, so the remainder of the block needs to have a single payload octet to associate with the extension blocks that come after the Payload Block.

169. **Provide functionality to link all the fragments of a bundle into a set identified by the *Fragment Identifier***, whether created locally or received from a link and maintain state about the fragments of a bundle known to the BD. *Fragments* linked to a single *Fragment Identifier* and hence being *fragments* of a single bundle are stored in order of fragment offset in the *bundle*. This makes it simpler to determine whether the currently available *fragments* are a 'covering set' for the *bundle*, i.e., the entire *payload* and hence the entire *bundle* can be reconstructed from the *fragment* set.

170. Provide functionality to delete the fragmentation state created at item 169.

171. **Provide functionality to reassemble a complete bundle from fragments at destinations**. This is primarily done at the bundle destination before delivery can be effected. At intermediate nodes fragments are normally left as they are received and forwarded as fragments. This avoids having to wait for all the fragments to arrive at the intermediate node, which might never happen, and avoids sillinesses where a bundle is reassembled only to be immediately refragmented. However, it is potentially possible that it might be necessary to reassemble a bundle at an intermediate *security-destination* (see [BSP]) if some security ciphersuite has to be unwound at this point. Although this is a theoretical possibility, there are some issues that would need to be sorted out with routing and forwarding to ensure that all the fragments actually arrived at the security destination. It is probably necessary to set the 'Do not fragment' flag at present t ensure that intermediate security destinations will work. [DTN2 does not handle reassembly other than at the final destination.]

4.6.5 Bundle Custody and Custody Timers

As part of the Store, Carry and Forward paradigm of DTN, a BD can offer to take custody of a bundle should the bundle request it and if the BD is configured to provide custodial services. Offering custodial services for a bundle implies that there has to be commitment to provide reliable delivery of the bundle both when it was despatched from its original source and in a node that is offering to take custody. Taking custody of a bundle means that this BD contracts to take over responsibility for safe holding the bundle until it can be successfully delivered to a new custodian further along the path to the destination or delivered to that destination. There is no point in taking custody of a bundle if the original sender was not attempting to achieve reliable delivery or if the new custodian cannot offer support for onward reliable delivery and robust persistent storage until that delivery can be verified.

Since it will generally not be possible to provide a low latency transaction type scheme to implement reliable delivery in a DTN environment, reliability involves explicit notifications, timeouts and retransmissions. Notifications take the form of *custody signals* (administrative bundles reporting successful and unsuccessful custody transfer). *Custody timeouts* are used to ensure that expected custody signals do actually arrive, and to trigger retransmission of bundles by updating the forwarding log (see Section 4.6.6) of the bundle to inform the router that it needs to forward the bundle again.

Whenever a bundle for which this BD has custody is forwarded on a link, a *custody timer* is started for that *(bundle, link)* combination. The parameters used for the timer have a set of global defaults for this BD that can be configured through the management interface (see items 430, 431 and 432). Where routes are configured through the management interface, it is possible to set alternative parameters for the custody timeout parameters on a per route basis (see items 448 and 449). [However, in DTN2, as with route priorities only the parameters for the *next-hop routes* will actually be used - values for *waypoint routes* will be ignored (although in practice they might be more appropriate as they may say something about the more distant waypoints which might actually be custodians whereas the next-hop might not be). In most cases *next-hop routes* are created dynamically in responses to link discovery, and these links always use the current global default values for custody timeouts.]

The custody timeout used for a bundle is not just a fixed value. The calculation is based on three parameters:

- a minimum timeout,
- a percentage of the bundle lifetime (expiration period), and
- a maximum timeout.

When a custody timer is created, the timeout is calculated as shown at item 173 below. The custody timer records the (bundle, link) pair to which the timeout applies.

The following functionality is provided in connection with custody timers:

172. **Set the global defaults for the custody timeout parameters from option strings**.

173. **Calculate the bundle custody timeout value**:
timeout = minimum (maximum timeout, (minimum timeout + (bundle lifetime * percentage))).

174. **If a custody timer times out, post a *Custody Timeout* event on the bundle core (see item 98).** Pass in the relevant *bundle* and *link* as parameters.

4.6.6 Forwarding Information and the Forwarding Log

A *Forwarding Log* is maintained for each bundle to record the history of a bundle as it is first either received as an incoming bundle from another node or created by an internal application or external (user) application on this node, and then delivered to an application on this node or forwarded on one or more links (possibly both of these). As well as 'successes', the Forwarding Log also needs to record some other items which could be viewed as 'failures' including custody timeouts and transmission failures.

The record of operations that have happened to a bundle is used by *routers* to help with deciding whether a bundle should be forwarded over a *link*. It allows a router to avoid naïvely resending a bundle to a given next-hop multiple times and to know when a resend *is* needed because the custody timeout has expired requiring a retransmission or because the link failed before a scheduled transmission could occur. To assist, with this the forwarding log for a bundle is recorded as a list of events in chronological order and can be searched so as to find the most recent event associated with a given link or next-hop EID.

In order to allow shutdown and restart of the BD, the Forwarding Log for a bundle needs to be part of the state for the bundle stored in persistent storage. It is the most variable part of the bundle as it is added to when a new operation is performed on the bundle. The changes are, however, always additions to the log - existing entries are not normally modified. Because the forwarding log is maintained in terms of links and remote EIDs, a naming system is needed for links that is persistent across BD shutdowns and restarts. Also the forwarding log entries themselves need to have a unique identification in the current persistent state database.

The forwarding information includes the following items:

- The unique identifier of the forwarding log entry.

- The action associated with this entry: either FORWARD action - send to just one link - or COPY action - send a copy of the bundle to several links.
- The type of log entry :
  - NONE - indicates no entry in log as a return value.
  - QUEUED - currently in the *send queue* or *inflight queue* of a link.
  - TRANSMITTED - has been successfully sent on the link indicated.
  - TRANSMIT_FAILED - transmit on the link did not succeed.
  - CANCELLED - transmission on the link was cancelled.
  - CUSTODY_TIMEOUT - the custody timer for the (bundle, link) pair expired before a *custody signal* was received for a transmitted bundle.
  - DELIVERED - the bundle was delivered to a local registration
  - SUPPRESSED - transmission was suppressed
  - RECEIVED - records the origin of the bundle,
- The name of the link on which the bundle was forwarded (if it was forwarded or forwarding is in process).
- The *registration ID* to which it was delivered if it was delivered.
- The *remote EID* to which the bundle was forwarded.
- The *timestamp* when the log entry was created or updated.
- The *custody timer specification* for this *(bundle, link)* pair

Forwarding information log entries must be capable of being written to and retrieved from persistent storage.

[In DTN2 forwarding log entries which refer to transmitted bundles record the custody timeout parameters for the link (see Section 4.6.5). This in principle makes it easier to set the correct parameters when the bundle is retransmitted. As it stands at the moment, the forwarding log is modified rather than having a new entry added so that the custody timer information is preserved for when the bundle is retransmitted.. This is not nice IMO. Also the forwarding log is not currently written to persistent storage: this is primarily due to the lack of persistent names for links.]

4.7 ENDPOINT IDENTIFIER MANAGEMENT

As documented in [URIscheme], bundles are routed to their destinations using locators known as 'Endpoint Identifiers' (Endpoint ID or EID). EIDs take the form of a Uniform Resource Identifier (URI) [RFC3986]. In principle EIDs could be taken from any URI *scheme* but the default scheme used in DTN2 and specified in [URIscheme] is *dtn:*. The part of the URI to the right of the scheme identifier (e.g., dtn:) is known as the '*scheme specific part*' (SSP).

In the existing DTN2 the first component of the SSP in the dtn: scheme (in URI parlance the *authority*) must take the form of a DNS compatible name (e.g., //dtnnode.example.com) or the whole SSP must be 'none'. The remainder of the SSP, after the next '/', in the scheme specific part is known as the *service tag* and is used to demultiplex bundles destined for the node delivering them to the application that has registered the *service tag*. In URI parlance the *service tag* is a combination of the *path* component and anything that comes after such as *query* and *fragment* components of the URI.

The proposals in [URIscheme] and [URIfind] describe much richer semantics for a standardized *dtn:* URI scheme, but are still under discussion. This specification describes the basic functionality used in DTN2 pending agreement of the improved scheme. In particular, the improved scheme would allow a BD node to be identified by several different EIDs that have alternative authority components, with one of these identified as the master EID used as the destination for custody signals and for handling other administrative tasks.

In the meantime:

175. **A node must have a distinguished Local EID (LEID)**.

176. **Bundles with the LEID as destination EID will be delivered to the internal administrative application in the BD**. Such bundles are normally expected to contain an administrative record such as a custody report that is meaningful to the BD.

4.7.1 Parsing Endpoint Identifiers

In DTN2, *dtn:* scheme EIDs are 'simple' URIs, similar to the URLs used in web browsers with a single ':'. The components after the 'hostname-like' authority component (after the third '/') are used for demultiplexing bundles to *registrations* supplied by applications. Registrations may contain one or more instances of the wildcard character '*' indicating that the registration EID should be treated as a limited form of pattern (akin to the basic 'globbing' scheme used in *nix shells to do filename matching) so that a set of EIDs will match the registration. Future versions of the Bundle Daemon will need to support more complex EID schemes as documented in [DTN-URI] and [DTN-FIND] where the SSP is able to specify the node and target application in more sophisticated ways. [TODO: Understand the SINGLETON/non-SINGLETON (MULTINODE) distinction for EIDs. DTN2 does not really handle MULTINODE EIDs at present.]

177. **The LEID can be set and recorded**. It will be written to persistent storage as 'global data'.

178. **Multiple URI schemes can be used**.

179. **The dtn: URI scheme is supported**. Other URI schemes can be supported if configured into the BD at build time. [DTN2 supports three non-trivial additional schemes as well as the trivial **String** (str:<string>) and **Wildcard** ("*") schemes :
    - **Ethernet** (eth://<Numerical MAC address>/<service tag>): Used with the Ethernet convergence layer to route bundles using the Ethernet link layer directly.
    - **TCA** (tca://<router id>/<service tag>: A prototype scheme using global identifiers to direct bundles to service suppliers such as email accounts.
    - **Session** (dtn-session://<node id>/<service tag>): Used for the publish-subscribe scheme.].

180. **For each URI scheme supported**, methods will be provided to
    180.1 create an EID object according in a supported scheme from a string representing the SSP,
    180.2 validate the syntax and semantics of the SSP of an alleged EID provided by an application or incoming bundle according to the specified supported scheme

180.3 perform a pattern match on the SSPs of two EIDs from the same scheme taking into account potential wildcard characters in both SSPs, reporting match or no match
180.4 create a new EID in a supported scheme by extending the SSP with an additional service tag or wildcard.
180.5 generate a string representation of an EID according to its scheme.

4.8 BUNDLE ROUTER

The *Bundle Router* is the main decision maker for all routing and forwarding decisions related to bundles.

It receives events from the *Bundle Daemon Core* after they have been processed in the bundle core having been posted by other components. These events include all operations and occurrences that may affect bundle delivery, including new bundle arrival, contact opening and closing, timeouts, etc.

To support the implementation of different routing protocols and frameworks, various different router components have been implemented conforming to the Bundle Router model. In DTN2 at present only one router may be active in a BD instance. Thus means that all links and contacts connecting to this BD have to use the same routing protocol and framework. If a BD using a different router mechanism connects to this BD there is no guarantee that bundles will be forwarded satisfactorily. [DTN2 does not currently have a way to verify that a connected BD is using a compatible routing scheme. It also is unable to act as the interface between sections of the network that might wish to use different routing protocols.]

Routing mechanisms can be classified into static and dynamic mechanisms. For static mechanisms the routes are worked out in advance either due to static configuration or a schedule. For dynamic mechanisms information (normally known as *metadata*)is exchanged between BDs that use it to determine routes and whether to forward bundles on particular links.

In response to network *link* opening events, timer events and the state of the links, the active *Bundle Router* determines which received bundles should to be sent on the available links. For dynamic routing the *Bundle Router* handles interchange of routing metadata with opportunistically connected nodes to provide guidance on the choice of bundles to forward (and to receive).

[DTN2 contains implementations for a number of routing protocols. Not all of the set are described in this functional specification. We describe only these that are proposed for use in N4C. These are static configured table based routing, epidemic routing and PRoPHET dynamic routing.]

4.8.1 Bundle Router Generic Functionality

Routers share a number of configuration variables as follows:

- The name of the type of routing algorithm currently active.

- Whether to add a route to the router for *next-hop* links where the remote EID of the link is known (defaults to true). Only relevant to table based routing mechanisms at present.
- Whether or not to open discovered opportunistic links as they become available (defaults to true). When a new neighbour is discovered it ultimately results in a *Link Available* event being posted with the reason flagged as *contact discovered*. This is processed by the router which can decide if the link should be opened immediately.
- Default priority for new routes. It defaults to zero. It is used to set the priority for new route entries. [AFAICS none of the extant DTN2 routers use this capability.]
- Table based routing can follow chains of routes to find a suitable next-hop on which to send a bundle. The search depth is limited by a parameter that sets the maximum number of links that should be followed (defaults to 10).
- Storage quota for bundle payloads. Defaults to unlimited. If the storage quota is (or would be exceeded) incoming bundles should be rejected.
- Subscription timeout. [Special for publish/subscribe mechanism - not needed for N4C].

Values for these configuration variables can be set by configuration commands as described in Section 4.11.7.11 and, for the payload storage quota, item 495.

The router also has a name (set when the router is created), and maintains convenience links to the pending bundles queue and custody bundles list created by the bundle core. In principle the active router will be offered all the events posted on the bundle core after the bundle core has finished processing the event. However the bundle core can decide that the router does not need to process certain events, so not all events will come down to the router. In many cases events are not propagated because they relate to an instance that is in process of being deleted and sending the event on would be confusing and potentially dangerous. Not all events will be interesting to the router. Typically a router would be interested in events such as:

- Bundle received.
- Bundle transmitted.
- Bundle cancelled.
- Route add.
- Route delete.
- Contact up.
- Contact down.
- Link available.
- Link created.
- Link deleted.
- Custody timeout.
- Registration added
- Registration removed.
- Registration expired.
- Shutdown request

Event handlers for the events that a router is interested in are implemented in each specialized router.

DTN Bundle Routers can either be designed to work by sending a single copy of a bundle along a preferred route or by broadcasting several copies of a bundle across multiple routes depending on what routes appear to offer the best probability of the bundle reaching its destination. To support these alternatives, routes used by the Bundle Router are associated with a *forwarding action* type which can be either FORWARD_ACTION for the single copy mode or COPY_ACTION for the multicast mode.

The following additional generic functionality is provided:

181. **Router factory function**. Called once during start up to create the router used by this BD. The router type to be used is specified by a configuration command (see item 441). This command should be used in the BD configuration file (see item 13). Sets default values for the parameters described in this section and copies in convenience links to the pending bundles queue and the custody list from the bundle core.

182. **Check if a bundle should be forwarded on a specified link given a specific action type**. It shouldn't be if the last forwarding log entry for the bundle on this link records that the last action was to queue it or transmit it on this link. If the remote EID of the link is known, check if the bundle has just been sent via some other link to the same place. Don't forward if it has been. Likewise if transmission to the remote EID has been suppressed. [Suppression is not used by any of the routers that N4C uses. It is mainly used in the current version of DTN2 - 2.7.0 - by the DTLSR router for controlling link state advertisement propagation - see item 202. It is also used by the Publish/Subscribe mechanism to suppress forwarding of subscription bundles when an upstream node has already provided a subscription.] The *action type* supplied as a parameter may be either to forward the bundle only to the specified next-hop on the given link or to forward a copy of the bundle to this next-hop (as in *epidemic routing* - see Section 4.8.3.2). If the *action type* is to forward a copy, report that it should (always) be forwarded. Otherwise it is forwarded only if the destination EID is a singleton node.

183. **Determine if the BD will accept an incoming bundle**. Currently the only test that is made is to check that adding the payload to the payload store will not exceed the storage payload quota (see introduction to Section 4.8.1 and item 495).

184. **Check to see if a bundle can be deleted**. This routine may be specialized by specific routers. By default it just checks that the bundle only has one 'mapping' i.e., it is only on the bundle core *all bundles* list. This functionality is called from the *try to delete* bundle functionality in the bundle core (see item 118).

185. **Delete bundle**. Specialized in specific routers.

186. **Recompute routes**. Intended to be used from the management interface to force route recomputation, e.g., after adding or deleting routes from the static router. By default this does nothing. It may be specialized for specific routers.

4.8.2 Table-based Routing Model

The Table Based Routing model is used to support various types of routing where there is an underlying link topology that can be captured as a table of routes. Unlike the

topologies of the conventional Internet, the topology can be potential rather than requiring that all the links in the topology are active and usable at the same time. The DTN *store, carry, forward* paradigm will cover situations where certain links in the topology are not active at all times.

The setup of routes can be managed by configuration commands or by an exchange of topology information though a (bundle-based) protocol (e.g., Delay Tolerant Link State Routing - DTLSR). The routing state is stored in a table in the BD router. [Note that this version of the functional specification does not provide details of the functionality of DTLSR.] The Table-based Routing Model contrasts with both scheduled routing models and dynamic routing models where there is not a relatively stable underlying topology and the routing is dependent on opportunistic encounters and delivery probabilities.

Each entry in the routing table contains an endpoint ID pattern that is matched against the destination address in the various bundles to determine if the route entry should be used for the bundle. The route is determined either by a link to a next-hop which should be used to forward bundles towards the destination (a *next-hop* route) or the EID of a node which should be used as a *waypoint* in forwarding the bundle to the destination (*waypoint* route). In the second case, the waypoint may not be one DTN hop away, and calculating the link to use to forward the bundle requires a recursive look up of routes to the waypoint until a *next-hop* route is found (or the lookup depth is exceeded, probably indicating a loop in the route).

If the [BSP] is in use, providing *waypoint routes* would potentially be a way of triggering the use of intermediate *security destinations* (see Section 4.6.3). However the current implementation does not remember the *waypoints* that were found before the *next-hop* route is identified.

[The DTN2 header file describing route (table) entries states: "There is also a pointer to either an interface or a link for each entry. In case the entry contains a link, then that link will be used to send the bundle. If there is no link, there must be an interface. In that case, bundles which match the entry will cause the router to create a new link to the given endpoint whenever a bundle arrives that matches the route entry. This new link is then typically added to the route table." It appears that is out of date. Routes do not interact with interfaces at all now. A route either has an associated link or a waypoint EID to direct bundle forwarding. This is determined at creation time and never altered.]

An entry may have a source EID that is used to compare with the source of the bundle and limit the use of the route to bundles from certain sources. This defaults to any EID.

An entry also has a forwarding action type code which indicates whether the bundle should be forwarded to this next hop and others (COPY_ACTION) or sent only to the given next hop (FORWARD_ACTION). The entry also stores the custody transfer timeout parameters, unique for a given route.

Each route is also assigned a priority which is used to differentiate between multiple routes to the same remote EID should this be necessary and is associated with one or more of the bundle classes of service allowing (in principle) QoS routing.

The parameters mentioned above can be configured for each route entry using the standard options mechanism, if the routes are configured through the configuration interface (see Section 4.11.7.11.1, items 448 and 449)

Some link specific information pertaining to a router can be attached to a link (see Section 4.9.1.4).

The router maintains a *reception cache* of recently received bundle information to help it manage duplicates and avoid routing loops using reverse path forwarding checks (see items 187 and 201). The cache is indexed by *Global Bundle Or Fragment Identifier (GBOFID)*.

The router can run *reroute timers* that allow it to hold off rerouting due to a link that is (hopefully) transiently unavailable for forwarding. This may be due to a large bundle clogging up the transmission path or a large number of bundles in the *send queue* or a transiently failed communications link that will be reopened shortly.

The router information stored for a link consists of a *deferred list* that contains a list of bundles that can't be immediately queued for transmission because the link *send queue* is full. This is not quite as stupid as it seems at first sight. Putting the bundle onto the link *send queue* causes the bundle storage requirement to increase because the wire format for the bundle is generated at the point when the bundle is placed into the *send queue*, so the requirement for additional resources can be staved off by leaving the bundle in only its internal format until the *send queue* has drained somewhat.

[In principle, routes can be stored in persistent storage but this is not currently done in DTN2.]

[There is a good deal of specialized functionality for the publish/subscribe mechanisms. This is omitted from this version of the functional specification.]

4.8.2.1 Table Based Router Event Handlers

The following event handlers are provided:

187. **Bundle received**. The remote EID is obtained from the link specification on which the bundle was received if the remote EID was specified. Otherwise the remote EID is set to the null EID. [It should be possible to check if there was a *Previous Hop* Block in the bundle and use the EID in this block as the remote EID. DTN2 does not do this.] Add the bundle to the *reception cache*. This will be refused if the bundle is a duplicate. If it is a duplicate, post a *Bundle Delete Request* event on the bundle core (see item 66) and do no more. Assume initially that the bundle should be routed at the end of this functionality. If doing publish/subscribe, the bundle is examined to see if it is part of a session. If so add the bundle to this session in this BD. It may turn out that the bundle has already been obsoleted in which case it should not be routed and there is nothing more to do. It may also be a session subscription bundle. Handle the session subscription which will return whether the bundle should be routed; in some circumstances the result will be that further forwarding of a session subscription bundle is *suppressed* because this BD is already subscribed from an upstream node. If so determined *route the bundle* (see item

202), and otherwise post a *Bundle Delete Request* event on the bundle core (see item 66).

188. **Bundle Transmitted**. If the bundle has a deferred single copy transmission for forwarding on any links, then remove the bundle from the deferred lists associated with those links (see item 203). Check if the completed transmission has made room to move any bundles from the deferred list to the send queue for the link where the transmission has completed (see item 204).

189. **Bundle Send Cancelled**. If the bundle has not expired, it needs to be rerouted (see item 202).

190. **Route Add**. Perform the *add route* functionality (see item 207).

191. **Route Delete**. Perform the *delete route* functionality (see item 208).

192. **Contact Up**. Check the link is actually up and report an error if it isn't. Otherwise add the link as a *next-hop route* if so configured (see item 209). Then check if a bundle could actually be forwarded on this link and if so queue as many bundles from the *deferred queue* as possible for sending (see item 204). Cancel any pending *reroute timer* (see Section 4.8.2) for this link as the link is now reopened.

193. **Contact Down**. If there are any bundles queued on the link schedule a *reroute timer* (see Section 4.8.2) for the link. The period before the timer will expire is configured by the *potential downtime* parameter for the link (see item 251). Record the timer as associated with the link. When the timer expires it calls the *reroute bundles* functionality for the link (see item 210).

194. **Link Available**. If the link is an opportunistic link and it is not open and the reason for this event is that the link has just been discovered, then if the router is configured to open discovered links (see Section 4.8.1) call the bundle core functionality to *open the link* (see item 111) which will eventually result in a *Contact Up* event being posted. Check that bundles can now be forwarded on the link and cancel any existing reroute timers (see item 204). This appears redundant as the *Contact Up* event handler (see item 192) does it as well - but it does avoid a potential race condition in which the timer fires while the connection is being made.

195. **Link Created**. Create an empty *deferred bundles* queue (see Section 4.8.2) and store a pointer to it in the link data structure. Add a *next-hop route* for the link to the routing table (see item 209).

196. **Link Deleted**. Delete any entries in the routing table that refer to the link's next-hop address. Cancel any existing *reroute timers* (see Section 4.8.2) associated with the link.

197. **Custody Timeout**. The bundle needs to be forwarded again due to no *custody accepted* report having been received before the deadline implemented by the custody timer. Call the *route bundle* functionality (see item 202).

198. **Registration Added**. Deals with the publish/subscribe aspects of adding a *registration*. [Not relevant to N4C].

199. **Registration Removed**. No action required for publish/subscribe aspects of removing a *registration*. [Not relevant to N4C].

200. **Registration Expired**. Deals with the publish/subscribe aspects of expiring a *registration*. [Not relevant to N4C].

4.8.2.2 Table Based Router Supporting Functionality

The following pieces of functionality are provided [DTN2 provides additional functionality for the publish/subscribe mechanisms but this is not documented in this version.]:

201. **Check whether a bundle should be forwarded**. A reverse path forwarding (RPF) check is done to help avoid routing loops. The previous hop from which the bundle was received is checked by looking up the bundle in the *reception cache* (see Section 4.8.2). The bundle should not be forwarded if the next-hop on this route takes the bundle back to the node it came from. Note that the *reception cache* is cleared when routes change (see item 213); if routes have changed the bundle can potentially be sent back the way it came. If it is still OK to forward at this stage, also run the *should forward* functionality specified for the generic routing table (see item 182).

202. **Route a bundle**. Check if forwarding to all nodes is SUPPRESSED. [This capability is only used by the publish/subscribe mechanism for subscription bundles in Table Based Routers and for stale Link State Advertisements in DTLSR at present.] Create a list of *next-hop routes* that will (eventually) lead to the destination. This is done by scanning the routing table for routes where the route destination EID matches the bundle destination EID (with the match being carried out according to the scheme to which the EIDs belong. If a *waypoint route* is found, a recursive search is carried out to find a *next-hop route* that leads through the waypoint to the intended destination. The search depth is limited by the *chain length limit* parameter configured in the router (see Section 4.8.1). Multiple routes may exist. Sort the routes based on the priority assigned to the routes in the list - note that only the priority assigned to *next-hop* routes has any effect here - forwarding will be attempted on the highest priority next-hop route first. The default sort mechanism breaks ties by prioritizing links that have the fewest bytes already queued for forwarding. [DTN2 is designed so that the sorting algorithm can be overridden but this capability is not used as yet.] Iterate through the selected routes checking if the route's link is a good link to use (execute the *should* forward functionality - see item 201); if so, check that the bundle is not already in the link's *deferred list* and reject this link if it is. If the link is still a possibility at this stage, make sure that any bundles that could be are transferred from the *deferred list* to the *send queue* to avoid the bundle leapfrogging any bundles on the *deferred list* (see item 204). After all this forward the bundle to the next-hop on any acceptable link (see item 214), and count how many times it was forwarded. [I am not sure how this square with the *action type*: if it is FORWARD_ACTION we may still get forwarding on multiple links apparently..]

203. **Remove a specified bundle from all deferred lists that match a specific set of actions**. Scan all the active links recorded by the Contact Manager (see Section 4.9.8.1). If a link has no router information entry, skip it (this function may be called early on in startup before links have any router information). Otherwise scan the

bundles in the *deferred list* for the specified bundle and delete it from the deferred list if its *action set* is a subset of the action mask passed in.

204. **Check if a bundle could currently be sent on a link to the next hop specified by a route and move any deferred bundles that can be sent from the link's *deferred list* to its *send queue***. Check the link is open (i.e., has an active contact) and there is some space in the link's *send queue* (i.e., the count of bundles and octets in the *send queue* is below the *low water marks* set for the link). If these checks succeed it should be possible to send a bundle. Iterate through the link's *deferred list* of bundles. For each bundle check if the link's *send queue* is now full. If not, use the basic *should forward* functionality (see item 182) to check if the bundle has already been transmitted or is inflight on another link. If so the bundle should not be sent on this link. Otherwise if the link is available but not in the OPEN or OPENING state, open the link using the bundle core *open link* functionality (see item 111). Delete the bundle from the link's deferred list and queue the bundle in the link's send queue using the bundle core's *queue bundle* functionality (see item 114).

205. **Determine if the router agrees that a bundle proposed for possible early deletion can be deleted now (see item 118)**. Check if the bundle has either been forwarded or transmitted on any link by inspecting the bundle's forwarding log. If not then it cannot be deleted yet. Check if this BD has custody of the bundle. If so then it cannot be deleted yet. If the bundle is part of a publish/subscribe session with subscribers on this node then it cannot be deleted yet. Otherwise it can be deleted.

206. **Extra actions to be performed when deleting a bundle (see item 120) and the router is a Table based router**. Remove any instances of the bundle from the deferred lists of all links (see item 203). Remove the bundle from any session with which it is associated.

207. **Add a route to the routing table**. Add the new route entry to the route table. Consider rerouting bundles in the face of changed routes by calling *handle changed routes* (see item 213).

208. **Delete routes matching a destination EID**. Scan all the routes in the routing table and delete any that match the destination EID according to the scheme being used for the destination EID. Clear the *reception cache* (see Section 4.8.2). Consider rerouting bundles in the face of changed routes by calling *handle changed routes* (see item 213). [DTN2 does not do this but ought to according to a note in the code.]

209. **Add a *next-hop route***. If the router is configured to add next-hop routes when links are discovered (see Section 4.11.7.11 and item 239) and the remote EID of the new link is known (i.e., not the null EID), build a new route table entry from the remote EID address. If possible add a *service tag* wildcard to the remote EID. If this is not possible try to install a route for the basic remote EID. In both cases check that the *route entry* doesn't exist already. If not add a new entry with the forwarding *action type* set to provide the single copy forwarding action.

210. **Reroute bundles queued on a link**. Check that the link is actually UNAVAILABLE. If not log a warning and do no more. There should be at least one bundle queued for sending on the link. Iterate through the bundles in the link's *send queue* and call the bundle core *cancel bundle* functionality for each bundle (see Section 4.9.1.4 and item 115). This results in a *Bundle Send Cancelled* event being posted for each bundle in the send queue (see item 63). The router's event handler for this event (see item 189) will do the actual rerouting. Verify that the link *inflight* queue is empty as it should be since the link is UNAVAILABLE.

211. **Reroute all bundles**. Iterate through the *pending bundles* queue and execute the *route bundle* functionality for each bundle in turn (see item 202). If the routing decision has altered for any bundle, cancel the bundle from the previously selected link before queuing it on the new link. [This last piece of functionality is not implemented in DTN2 although there is a TODO note about it.]

212. **Recompute all routes**. Alias for reroute all bundles (see item 211).

213. **Handle changed routes**. Clear the *reception cache* (see Section 4.8.2) since we may now legitimately want to send bundles back where they came from. Then execute *reroute all bundles* and (for the publish/subscribe mechanism) *reroute all sessions* [not described at present].

214. **Forward bundle to the next-hop on the link specified in a routing table entry**. If the link is available but not (yet) open or being opened (in state OPEN or OPENING), open the link using the bundle core *open link* functionality (see item 111). If the link is open and the link's *send queue* is not full, use the bundle core's *queue bundle* functionality (see item 114) to place the bundle on the link's *send queue* and return that the operation was successful. Otherwise, if the bundle is not already on the link's *deferred list*, add it to the deferred list. For debugging purposes log a message indicating why the bundle was not (yet) forwarded . Return that the operation was not successful.

4.8.3 Example Table Based Routers

4.8.3.1 Static Routing

The Table Based Routing Model described in Section 4.8.2 can be used directly to provide a purely managed static routing system. All routes apart from discovered *next-hop routes* are created and destroyed by configuration commands (see Section 4.11.7.11).

4.8.3.2 Epidemic or Flood Routing

Epidemic or *Flood routing* does no work to select a specific link that is particularly appropriate to send a bundle on to have it reach its destination with minimal or optimal use of resources. Instead every bundle to be forwarded is sent once on every link that is open during its lifetime. For received bundles this should preferably not include the link on which it was received. [DTN2 currently does not implement this restriction. This is a waste of resources as it will be deleted as a duplicate when received by the node that sent it.]

Epidemic Routing is provided by some minor modifications to the Table Based Routing model (see Section 4.8.2). The Epidemic Router maintains an *(epidemic) all bundles* list

that holds a reference to each bundle currently held in the BD. (Note: this is a different, additional list to the *all bundles* list maintained by the bundle core which is used to manage garbage collection and persistent storage of bundles as introduced in item 32.) The router is configured so that *next-hop routes* are not automatically added when a link is discovered (see Section 4.8.1). Instead a 'wildcard' route is added whenever a link is created that will match the destination EID of any bundle to is received or generated. The route is created with the COPY_ACTION forwarding action so that the bundle is sent on all available routes rather than just the most preferred. All these routes have equal priorities. The normal Table based Router model forwarding will carry out the forwarding as required with this set of routes, sending one copy of the bundle on each link as it becomes available until the bundle expires. The *can delete bundle* functionality (see items 118, 205 and 218) is modified so that the Epidemic Router always refuses to allow a bundle to be deleted before its expiry time arrives.

The following Table Based Router functionality is specialized for an Epidemic Router:

215. **Initialize the router to suppress the default creation of *next-hop routes* whenever a link is discovered** (see Section 4.8.1).

216. **Bundle Received event handler** (extends item 187). Add the bundle to the Epidemic Router *all bundles list*. Then execute the standard Table based Router functionality (see item 187).

217. **Link Created event handler** (replaces item 195). Create a new *route table entry* for the link with the '*wildcard' remote EID* as destination. Set the action for this route to COPY_ACTION. Add the route to the route table (see item 207). This will cause any existing bundles to be queued for forwarding on the new link.

218. **Determine if the router agrees that a bundle proposed for possible early deletion can be deleted now** (see item 118 - this replaces item 205). Always refuse to allow early deletion.

219. **Extra actions on deleting a bundle** (extends item 206). Delete the bundle from the Epidemic Router *all bundles* list. Then execute the standard Table based Router actions (see item 206). [DTN2 just deletes the bundle from the *all bundles* list. This potentially leaves it queued on some links' *deferred bundles list*.]

4.8.4 Dynamic Routing Manager

Only one dynamic routing algorithm has been implemented for practical use so far - the PRoPHET algorithm described in the next section.

4.8.5 PRoPHET Routing

PRoPHET Routing is described in **Probabilistic Routing Protocol for Intermittently Connected Networks** [PRoPHET]. This document gives a fairly complete expression of the functionality required to implement the PRoPHET routing protocol. PRoPHET is a dynamic routing protocol. This means that decisions on forwarding are not controlled by a table that has been compiled prior to an opportunity to forward bundles becoming available. Instead, a table of *delivery predictabilities* is maintained that gives the *delivery predictability* for known destinations.

When a new neighbour is discovered and communication established, routing *metadata* is exchanged which is used to inform the BD what *delivery predictabilities* the contact is using, so that each of them can decide if it is appropriate to copy bundles to the neighbour because this would give the bundle a better chance of delivery than just retaining the bundle in its current location. The delivery probabilities exchanged are then used to update the delivery predictabilities in both neighbours in accordance with the algorithms defined in Section 3.3 of the PRoPHET specification [PRoPHET]. Comparison of the delivery predictabilities between the current node and the neighbour correspondent node together with some additional policy information (such as how recently the bundle has already been forwarded) is used to generate an ordering of the set of bundles in the *pending queue* of each peer's BD. The result is used to make an offer of sending to the correspondent which prioritizes the bundles that would appear to have the best chance of delivery if sent to this correspondent for onward forwarding or local delivery - clearly any bundles actually destined for the correspondent will have a high priority!

The correspondent examines the offers and determines, based on the existing stock of bundles that it has and local resource constraints, which offers it proposes to accept. This pruned offer list is sent back to the originator which then commences sending the bundles.

A bundle can be sent multiple times to various next hops until either notification of successful delivery is received or the bundle expires. In essence, PRoPHET operates a form of epidemic routing but endeavours to prune the set of neighbours to which the bundle is sent to minimize the number of 'wasted' sends, where the bundle is sent off via a node that is unlikely to encounter other node that offer a good probability of getting the bundle to its destination.

[The DTN2 implementation of PRoPHET in version 2.7.0 does not match the specification that will be submitted as an experimental standard in 2010. It also does not conform to the coding conventions used in the remainder of the DTN2 reference implementation. One major divergence from the proposed experimental standard is that the routing metadata is exchanged in bundles over the same link as data bundles without being able to control prioritization of metadata. The draft of the standard suggests that metadata is exchanged over a separate connection to avoid interference. Accordingly this functional specification does not give the details of that implementation. A revised implementation is planned. The following outline suggests how this may be structured but this is still work in progress.]

PRoPHET assumes that a reliable local link transport exists and explicitly calls out TCP as a possible underlying network transport. It would be reasonable to assume that bundle exchange takes place using the same network transport as is used for the metadata exchange. We propose to derive a PRoPHET-TCP convergence layer inheriting the TCP link functionality and adding a TCP based metadata exchange. A PRoPHET-TCP convergence layer would listen on two TCP sockets: a conventional TCP convergence layer listener for bundle connections and an extra listener for metadata connections. Further this could derive from an extension to connection oriented convergence layers where the connection has an auxiliary metadata connection. The following functionality is envisioned:

220. **PRoPHET Discovery**. The standard IP and potentially Bluetooth discovery agents can handle PRoPHET discovery. The connection announced would be for the metadata exchange. This in turn would inform the remote peer about the bundle connection to use. This connection would be opened when the PRoPHET connection was established.

221. **The metadata exchange includes the EID of the remote node so that an existing link or new link could be activated to handle the bundle exchange**. The router information in the link (see Sections 4.9.1.4 and 4.9.4.1) maintains the list of delivery predictabilities for other EIDs associated with the remote EID for this link.

222. **A PRoPHET router maintains information about each bundle to support the bundle offer processing**. This includes the number of times that it has been forwarded and the highest delivery predictability that was discovered in a correspondent node. [DTN2 maintains this information is a separate PRoPHET all bundles list, but it should probably be maintained in the Forwarding Log.]

223. **A PRoPHET router maintains a set of *delivery predictabilities* for remote EIDs that it is aware of**. These are manipulated as specified in the [PRoPHET] specification when encountering other nodes.

224. **The PRoPHET router mediates bundle exchange with the correspondent.** After completing the data exchange, the agreed set of bundles for exchange are enqueued on the link *send queue* in the order agreed during the PRoPHET metadata exchange. If this is too much for the *send queue*, the *deferred bundles* mechanism used in Table Based routers can be used to hold bundles to be sent but without creating their wire format (saving space and processor resources). This mechanism could also be used if the expected duration of the contact is known. Only as many bundles as may be expected to be sent can be placed on the *send queue* with the *deferred bundles* providing a reserve if the contact goes on longer than expected.

225. **Dealing with late arriving bundles**. If additional bundles arrive that might need to be sent, additional metadata exchanges can be used to offer the extra bundle(s).

226. **Dealing with loss of connection**. Both metadata and bundle connections are closed if either connection loses connection or is explicitly closed.

4.9 CONTACT MANAGER

The *contact manager* is responsible for managing communications *links* from the local node to other nodes over which bundles can be sent. These *links* (see Section 4.9.1.4 for more detail) provide a way to send bundles to or receive bundles from a *next-hop* on the forwarding path. The next-hop node is not necessarily a bundle destination but it will have an active bundle daemon (BD) with associated EID(s).

Each link has an associated *convergence layer* which determines the low level transport protocol and link maintenance protocol used to support the exchange of bundles across the link (see Sections 4.9.1.2 and 4.9.5).

A link represents a potential or actual connection to a next hop. When there is an active connection a *contact* encapsulates all the information about the link and the data being

exchanged with the remote endpoint of the connection. The contact manager maintains the various pieces of state necessary to manage the contact. The contact manager can either work with configured 'managed' links or can use a *discovery* mechanism to advertise its own interface/convergence layer combinations to prospective communication partners and establish links in response to advertisements received from other nodes (see Section 4.9.3).

Links are categorized in various varieties that are characterized by the degree of persistence and the reason for opening of contacts on the link. Four varieties are defined: *always on*, *on demand*, *opportunistic*, and *scheduled*. The names are reasonably self-explanatory for the time being - they are fully defined in Section 4.9.1.4.

Links can be created in various ways:

- Administrative management request either from an initial configuration file or during operation.
- As a result of a advertisement received by a neighbour discovery mechanism.
- As a result of a connection request received on a passive communication *interface* configured for a convergence layer that handles connection oriented transport protocols (see Sections 4.9.1.1 and 4.9.2).

4.9.1 Components

4.9.1.1 Convergence Layers

The Bundle Protocol [RFC5050] does not specify exactly how bundles will be transported between nodes that support Bundle Protocol Agents (i.e., Bundle Daemons). Instead it expects that many different transport protocols will be used depending on the kind of network in which a node is situated. For each transport protocol that could be used an adaptation or *convergence layer* has to be defined that will allow bundles to be sent and received across the type of network on which the transport protocol runs (e.g., the TCP and UDP convergence layers allow bundles to be sent across an IP-based network using unicast transports, the Bluetooth convergence layer allows bundles to be sent between paired Bluetooth nodes, and the Ethernet convergence layer allows bundles to be sent between nodes in 'raw' Ethernet frames).

Convergence layers are named. Each type is implicitly associated with a network type and an implementation of a convergence layer is associated with the implementation of the network communication endpoints for that type of network on the platform where the implementation is deployed (e.g., sockets for IP networks).

Convergence layers may be associated with *bidirectional* or *unidirectional* communication endpoints. A link that uses a bidirectional communication endpoint (e.g., TCP over IP) would expect to be able to both send and receive bundles via the same endpoint. A convergence layer that uses unidirectional endpoints has to specify if the link endpoint is intended to send or receive bundles. It is possible that in this case a link maybe effectively unidirectional. [DTN2 does not explicitly support unidirectional links. An open *link* (i.e., one on which communication is established) is expected to be able to send and receive bundles.]

Convergence layers may offer reliable, acknowledged transport of the bundle (normally on bidirectional communication endpoints).

The *convergence layer* maintains state about each *link* and, if the *link* is active, the *contact* that is using that link and convergence layer. The convergence layer provides all the functions needed to control the communication endpoints used by the convergence layer, establish and maintain transport connections using the selected protocol, and to send and receive bundles across established transport connections. The convergence layer accumulates statistics for each contact, and potentially uses these statistics to provide input to the routing system to influence the selection of forwarding path for subsequent bundles.

The convergence layer may offer a '*keepalive*' mechanism to allow the BD to verify that the link remains open. If no response is received after a number of repeats of the *keepalive* message, the connection will be assumed to have ceased and the link will be closed.

4.9.1.2 Types of Convergence Layer

Depending on the underlying transport mechanism, *convergence layers* can be categorized and the required functionality factored into common pieces and transport specific pieces to minimize the amount of duplication of work between convergence layers that use transport protocols with similar semantics.

For the purposes of eliminating common code, the major categorization splits convergence layers into *connection oriented* and *non-connection oriented* groups. There are significant differences between the ways in which the two types operate.

4.9.1.2.1 Non-Connection Oriented Convergence Layers

In this category, the transport protocol is expected to send and receive data in essentially independent packets or frames (*protocol data units* - PDUs) with no state maintained regarding the next-hop source or destination for the PDUs in the transport layer. Typical simple transport protocols used in DTN convergence layers are UDP/IP and Ethernet frames. More complex examples include the Licklider Transmission Protocol (LTP) [RFC5325] and the NACK-Oriented Reliable Multicast (NORM) convergence layers. With these kinds of protocol, the convergence layer has to be responsible for reliability (if available) and maintenance of (virtual) connections to provide the DTN link functionality needed. In some cases the transport layer endpoints may be effectively unidirectional, i.e., bundles can be sent or received but not both. For example, LTP is unidirectional. [How is this presented above the convergence layer in DTN2 - look at LTP when it arrives!] In other cases, such as UDP or Ethernet, endpoints can act as both senders and receivers.

For these convergence layers, new links are either created by local management action or by a separate discovery mechanism and are primarily intended for sending bundles. The transport protocols generally do not provide for opportunistic link creation. In this case interfaces provide the route for incoming bundles.

A special case of this category of convergence layer is the *Null Convergence Layer*. This is effectively a test tool which 'consumes' bundles and discards them. It has no way of receiving bundles. It is roughly similar to */dev/null*, often known as the 'bit bucket', on Unix systems.

4.9.1.2.2 Connection Oriented Convergence Layers

Transport layers that provide a managed connection between two endpoints (usually) on separate nodes can be conveniently used to transfer bundles to and receive bundles from a next-hop peer. Connections map one-to-one onto contacts for this type of convergence layer. Protocols such as TCP and SCTP in the IP domain, the Bluetooth Radio Frequency Communication (RFCOMM) transport and a simple point-to-point serial link using RS-232 or similar can be used for *connection oriented convergence layers*. This type of convergence layer may have to deal with the establishment of opportunistic links through connection requests from the transport layer (e.g., as with the *accept* socket API call in IP).

All the protocols mentioned above provide reliable [one could question whether the DTN2 Serial Convergence layer is really reliable, but we will gloss over that], in-order delivery of a undifferentiated stream of data - *stream convergence layers*. Generally the interface to such transports provides for the data to be passed to the protocol as a sequence of segments with a user defined size. The common functionality provides for a bundle to be split into segments of a configurable size that are then sent sequentially and similarly received at the remote end where they can be reconstructed into a bundle. Only a single bundle will be *inflight* on a connection at one time (i.e., we don't interleave segments from different bundles on the wire). Connections are expected to be bidirectional. The stream convergence layer can be configured to provide per segment acknowledgements so that the transmitting node can be certain that the receiving node has received an intact segment. In this case, the stream convergence layer has to maintain state on both sending and receiving side about which segments have been acknowledged, but since the underlying transports are reliable, it will not be necessary to provide retransmission capability in the convergence layer. However, if a connection closes unexpectedly during bundle transmission, the BD will have to either fragment the bundle that was inflight, if fragmentation is enabled and acknowledgements were being sent, or resend the whole bundle if it can't be reactively fragmented.

Stream convergence layers can also exchange *keepalive messages* to ensure that the connection remains open and functional. In the case of *on demand links*, a configurable idle timer is restarted whenever a bundle is sent or received. If the timer expires, the connection will be closed by sending an explicit shutdown message as the link is deemed to be idle. It will be automatically reopened if a new bundle has to be sent. If the connection is broken other than on request of the BD, *always on* and *on demand* links will be restablished, if possible, when a periodic timer operated by the contact manager expires next.

Sending of acknowledgements is prioritized over sending of bundle data, but will not interrupt the sending of a bundle segment in case the connection blocks before the complete bundle segment is sent, and the remainder of the segment has to be sent via a separate call to the write API.

On the receiving side, it is essential to keep the incoming side of the connection drained to ensure that the sending side does not deadlock waiting for acknowledgments.

Links using *connection oriented convergence layers* can either be created *actively* from this bundle daemon by requesting a connection with the next-hop remote peer or *passively* using a *interface* that listens for connections from remote peers. If a discovery mechanism

is in use in both potential peers of a connection, it will be a matter of chance as to which end point takes the passive role and which takes the active role. It is possible that connections may be initiated from both ends if the advertisements are fortuitously received at nearly the same time at both ends. The BD may wish to prune one or other of such contacts to avoid uncertainty. The discovery mechanism may be constrained if the path that the underlying transport takes lies through one or more network address translator (NAT) boxes or firewalls. In this case discovery will usually only work with advertisements sent from inside to outside, and may be dependent on the advertisement providing state in the NAT or firewall that will allow the reverse advertisement to pass through the firewall.

4.9.1.3 Interfaces

An *Interface* abstracts a bundle transport communications end point. Interfaces are named and associated with a specific *Convergence Layer* . Interfaces are not *a priori* associated with a network type or endpoint type: this is determined by the Convergence Layer using the interface. A Convergence Layer may be associated with multiple Interfaces if appropriate because the node has multiple physical or virtual network interfaces.

An active Interface (described as *up*) will be set up to receive data using the communication endpoint type (e.g., IP socket) and protocol appropriate to the convergence layer that it is using. The functionality of an interface depends on the type of convergence layer involved. For non-connection oriented convergence layers (see Section 4.9.1.2.1) the interface provides a route for receiving bundles. Incoming bundles received on an interface are not associated with a specific link and it may or may not be possible to identify the EID of the previous hop that sent the bundle. For connection oriented convergence layers (see Section 4.9.1.2.2) the interface provides a passive listener that allows connections to be requested by a remote peer resulting in the creation of an opportunistic link (see Section 4.9.1.4).

Interfaces are created and activated through administrative action, typically by initial configuration of the BD. There is no way to dynamically create interfaces other than by management commands. In principle it is possible to destroy an interface (described as taking the interface down) but this is not a terribly useful thing to do in most situations, except when the BD is terminating.

4.9.1.4 Links

A *link* represents a potential or actual contact with a remote peer over a specified convergence layer and using a specific remote address. The remote peer will be either the *next-hop node* for bundles being sent from this BD or the *previous-hop node* for bundles received from the remote peer.

Links are the key components that bring together the communication mechanisms used by DTN (the convergence layers), the data to be sent or received (the bundles) and the strategy and policy for forwarding the data (the router(s)). Consequently the link has to maintain a complex web of state information that ties the link to each of these other components and, in turn, provides the other components with the link-specific information that they need to manage their interactions with the link. Particular care has to be taken when unravelling these connections if a link is to be destroyed.

Links can be created by various means: as a result of management or configuration action in the local node, as a result of response to an *advertisement* from another node received by the *discovery* manager (see Section 4.9.2), or as a result of a connection request received on an interface for a connection oriented convergence layer (see Section 4.9.1.2.1).

Links are characterized by the degree of persistence that the connection exhibits. The following types of link are supported:

- *Always-on:* Configured links that are maintained in an open condition for as long as the partner node is accessible using the specified convergence layer through the specified interface.
- *On Demand:* Configured links that are opened when the bundle forwarder determines that there is a bundle to be sent on the link. The link will remain open unless no bundle is sent for an extended period of time; after the link has remained idle for a specified period the link will be closed to reduce the resources used by the BD, but will be reopened, if possible, if a new bundle has to be sent. [There is a comment in the DTN2 OpportunisticLink.h file that indicates that On Demand links 'do not have a queue of their own'. Presumably this would be the queue of bundles awaiting transmission. I can find no evidence that this is true or why it should be the case.]
- *Opportunistic:* Links that are opened in response to the receipt of an advertisement by a node that has become active or come into communication range. The link is closed again when the correspondent node goes out of range or ceases to be active.
- *Scheduled Link:* Links that are opened and closed according to a predefined schedule. This type of link is particularly appropriate for inter-planetary communications where satellites and other space probes are available for communication with earth stations according to the visibility of the satellite and pre-arranged activity periods.

Links can be in a number of states:

- *UNAVAILABLE:* The link is closed and not able to be opened at the moment.
- *AVAILABLE:* The link is closed but is able to be opened, either because it is an *On Demand* link, or because an opportunistic peer node is in close proximity but no convergence layer session has yet been opened.
- *OPENING:* A convergence layer session is in the process of being established. This state is needed where 'opening' a connection in a connection oriented convergence layer is not completely under local control, but might require significant time to exchange data with the remote partner involving one or more network round trips. Blocking the BD thread while this happens would not be appropriate.
- *OPEN:* A convergence layer session has been established, and the link has capacity for a bundle to be sent on it. This may be because no bundle is actually being sent, or because the convergence layer can handle multiple simultaneous bundle transmissions.
- *BUSY:* The link is OPEN but currently has no capacity for sending bundles due to a transmission in progress or excessive backlog. [In DTN2 the BUSY state is implemented by setting queue high and low water marks for both number of bundles queued and total number of octets in the queued bundles. These limits are

(or should be) tested before queuing a new bundle. It appears that only the Table Based Router honours this contract.]

- *CLOSED:* Transient state used to indicate that the link is no longer available for communication and will transition to the idle AVAILABLE or UNAVAILABLE state when the Convergence Layer has shut down the transport link.

Each link has a *send queue* of bundles awaiting transmission, and an *inflight queue* of bundles that are being actively worked on by the link, i.e., that are being transmitted or may be waiting for acknowledgements if the convergence layer offers reliable transmission. If the supposedly reliable link closes during transmission then it may be possible to convert the portion of the bundle that has been transmitted and acknowledged into a fragment, splitting off the remainder of the bundle for transmission in a later opportunity. The intention is to produce equivalent fragments at sending and receiving sides - however a typical two way handshake will generally result in the source being able to guarantee that less has been sent than was actually received. An overlap in the data received between fragments is permitted but of course some capacity is wasted and the amount of overlap should be minimized.

Links sit at the core of the interactions between the components of the BD: to support these interactions, Links provide a set of opaque data pointers that can be used by the various components that work with links (convergence layer, contact and router) to store data needed in these components which is specific to the link. Care has to be taken to decouple this information when deleting contacts or changing the convergence layer or the router, and especially when deleting a link.

4.9.1.5 Contacts

A link that is in active communication with its remote endpoint (next-hop) has an associated *contact*. The contact manages the data receiving and transmitting functions for an active (open) link and acts as the interface between the bundle core and the routers and the convergence layer functions that work with the communications hardware. The Contact Manager keeps track of the current contacts dealing with routing events and directing bundles to and from the correct contact.

4.9.1.6 Discovery Agents

Discovery Agents provide a means to advertise the willingness of this BD to exchange bundles using a specific convergence layer, and to receive such adverts from potential next-hop peers. Generally Discovery Agents (sometimes called Neighbour Discovery Agents, although they are not necessarily very close!) will be instantiated as a result of management action typically from an initial configuration specification.

Discovery Agents are applicable in situations where there is the likelihood of, at least, local low latency, bidirectional communications but such communications with a next-hop node are likely to be only intermittently available. A typical situation in which a Discovery Agent is useful would be in a node that was mobile and nomadic with Wi-Fi IP capability. The node would from time to time come into communications range of another similarly equipped node. The Discovery Agent would allow the prospective communication partners to find out about the existence of nodes that support a bundle agent and learn

their capabilities, allowing them to initiate an opportunistic link over which they can exchange bundles.

Although it could be otherwise, it will be almost always appropriate for the Discovery Agent to use the same network type as the convergence layers that it is used to advertise. Thus TCP and UDP convergence layers are most appropriately advertised using an IP based discovery scheme; a Bluetooth convergence layer is most appropriately advertised using the built-in Bluetooth Inquiry scheme [Inquiry]. However the advent of Low Energy Bluetooth technology may make this a convenient mechanism for advertising any type of communication capability in an energy challenged environment, even if Bluetooth is not used to exchange the data.

4.9.2 Interface Manager

The *interface manager* maintains a table of Interfaces that are the low level communication endpoints for the BD.

227. **Provide a function for adding an interface to the table of interfaces**. This is used to configure the interfaces of the BD. The function takes the name of the interface, the name of the Convergence Layer adapter to be used and a collection of parameters that are passed to the Convergence Layer component (see item 393). These parameters are Convergence Layer dependent and allow the Convergence Layer to establish and open the appropriate communication endpoint (see item 270). The request will be refused if an interface with the same name already exists. If the Convergence Layer succeeds in opening the endpoint and configuring it as requested, the interface is added to the table of available interfaces with the associated Convergence Layer attachment. The interface specification should be saved in persistent storage to allow links to be restablished when the BD is restarted. The functionality of an interface depends on the type of convergence layer associated with the interface. For connection oriented interfaces, it is typically used to accept new incoming connections. For non-connection oriented interfaces bundles may be sent and received through the endpoint. [DTN2 does not save interface specifications in persistent storage. This is part of the reason why the bundle forwarding log is not saved in persistent storage.]

228. **Provide access function to allow other components to search the table of active interfaces** for a named interface and return the table entry giving the Convergence Layer instance associated with the interface.

229. **Provide a function to allow an existing interface to be deleted from the table of interfaces by name**. The Convergence Layer component is called to shutdown the communication endpoint (see item 271). The persistent storage entry for the interface is deleted also. [DTN2 doesn't store interfaces in persistent storage. In most use cases deleting an interface is highly exceptional. Interfaces are likely to be configured during start-up and the likelihood of interfaces being deleted is small except during testing.]

4.9.3 Discovery Agent Manager

Much like Interfaces, *Discovery Agents* are generally created by the management subsystem (see item 388), typically as a result of instructions in the initial configuration file (see item

13). Several types of discovery mechanism can be supported. Each is specific to a type of network infrastructure such as IP or Bluetooth. The Discovery Agent Manager maintains the set of active Discovery Agents and provides means for adding new ones and deleting ones that are no longer required.

A Discovery Agent can be configured to *advertise* (also described as *announcing*) one or more *convergence layers*. Typically these will be associated with the network type used for the announcements but this is not required. [DTN2 IP Discovery and Bluetooth Discovery Agents only currently advertise respectively IP (TCP and/or UDP) convergence layers and Bluetooth (RFCOMM) convergence layer. The (IP) Bonjour Discovery mechanism only advertises the TCP convergence layer service.]

An advertisement contains the name of the convergence layer to be used and address information that can be used to either send bundles towards for non-connection oriented convergence layers or to establish a connection for connection oriented convergence layers (see item 390). Advertisements are sent out periodically according to a configured interval. For some types of discovery (e.g., Bluetooth) it may be necessary to apply some randomization to this interval to avoid synchronization issues.

On receiving an advertisement from a prospective next-hop neighbor, the Discovery Agent will pass the advertisement to the Link manager (see Section 4.9.4), to determine if there is an existing reusable link (i.e., one that was previously connected to the remote EID specified but which is currently unavailable) or, if not, to create a new opportunistic link to handle communications. [DTN2 does not provide any policy control or authentication to determine if links should be established. There is a potential denial of service security issue here.]

4.9.3.1 Discovery Manager Functionality

The Discovery Manager maintains a table of active Discovery Agents. Discovery Agents are identified by a textual name provided when they are created that must be unique across all Discovery Agents in the BD. The following functionality is provided:

230. **Create a new discovery agent and add it to the table of active agents**. The agent is identified by name which must be unique. The type of discovery agent to be created is determined by the 'address family' or type of network subsystem to be used to announce the availability of DTN bundle agent service. Parameters appropriate to the type of discovery agent will be supplied and passed to the created agent (see item 388).

231. **Shutdown and delete an existing discovery agent from the table of active agents**.

232. **Globally shutdown all active discovery agents**.

233. **Find a discovery agent by name**.

4.9.3.2 Generic Discovery Agent

This section describes the common functionality shared by all Discovery Agents. The agent maintains a list of convergence layer announcements that are advertised by this agent. A Discovery Agent has a number of parameters:

- The address family for which it is configured (identifies the network type and mechanism it uses).
- A string interpretable using the address family as the address to which outbound advertisements will be sent.
- A string interpretable using the address family as the address at which inbound advertisements will be received.

The following functionality is provided:

234. **Factory mechanism for generating and configuring a new *Discovery Agent***.

235. **Create and add a new *announcement* to the agent**. *Announcements* are identified by name, which must be unique within the *discovery agent*. A list of parameters is passed to this routine, the first of which is the type of announcement (see item 390). The type must be the name of a *convergence layer* type and determines the kind of announcement that is created. If the type is verified as a convergence layer name, an appropriate announcement is created if announcements are implemented for the type of convergence layer. If an announcement is created it is added to the agent's list of announcements. Specific announcement types may need to be informed that a new announcement has been added to the list (see item 236).

236. **Handle new announcement**. By default this function does nothing but may be specialized for specific types of discovery agent if they need to take action when a new announcement is added.

237. **Remove an announcement identified by name from the list of announcements for this agent**.

238. **Find an announcement by name within this agent**.

239. **Handle a neighbour discovered**. When an *advertisement* is received by a *discovery agent* this function is called with parameters giving the *convergence layer* type, a string representation the remote peer's address in the format appropriate for that convergence layer and the EID of the remote peer. The convergence layer is verified and the Contact Manager is consulted to determine if a link to this remote EID already exists (see item 345). If not a new opportunistic link is created (see item 350). The link is transitioned to state AVAILABLE if it is in the UNAVAILABLE state by posting a *Link State Change Request* event on the bundle core asking for the Link to change to state AVAILABLE (see item 84) with the reason indicating that a new contact has been discovered. This will result in a *Link Available* event being generated. This event is processed by routers which can decide whether to open the link immediately (see Section 4.8.1).

4.9.3.3 Generic Announcement Functionality

Announcements are specific to a convergence layer and record the type of convergence layer to which they apply. They are identified by a textual name. The periodicity of the announcement is configurable (in milliseconds). The following functionality is provided:

240. **Factory function to create an announcement of a specific type and with a given name.** If the type of announcement is supported, the created announcement is configured by specific functionality for the type of announcement (see item 246).

4.9.3.4 Functionality for Specific Types of Discovery Agent

4.9.3.4.1 IP Discovery

IP Discovery sends advertisements as UDP datagrams, one advertisement per datagram. [DTN2 only supports IPv4 and not IPv6.] The datagram can be sent broadcast (in which case they will only propagate to nodes on the local IP subnetwork), multicast to a specific multicast address [DTNRG should obtain a dedicated group address for this purpose] or unicast if the discovery is intended to check out one particular node. If multicast is selected the multicast maximum hop count can be configured. *IP Discovery* defaults to using broadcast. The same port is used for incoming and outbound advertisements. It must be configured for each IP announcement since there may be more than one of them. Note that on many operating systems user processes are not allowed to send to the IPv4 broadcast address and it may be necessary to specifically configure the system to allow user processes to send to multicast addresses.

The following specialized functionality is provided for IP Discovery Agents:

241. **Configuration**. A separate discovery agent thread is created to handle both incoming and outgoing advertisements. A UDP socket is opened and configured for the receiving and potentially, sending of advertisements according to the address parameters supplied (see item 388). The addresses for incoming and outgoing advertisements are recorded in string form (see Section 4.9.3.2). The thread must be capable of being notified whenever a new announcement is added to the list of announcements supported by the agent. The discovery agent thread for this agent is started.

242. **Discovery Thread Run Loop**. The discovery agent thread opens the advertisement sending socket and then runs a continuous loop that periodically sends advertisements and reads the incoming discovery socket to watch for incoming advertisements. Establish a notifier that can interrupt the poll executed later if a new announcement is inserted. During each execution of the loop each associated announcement is checked for the next time it should send an advertisement (see item 246). If any should send an advertisement, format the advertisement and send it. This may require an alternative socket if the local address is different to the local address used for the discovery socket. While checking the various announcements, record the lowest interval until the next advertisement needs to be sent. Check if the discovery agent thread has been told to shutdown (see item 245). If so exit the loop and close the sending socket. Otherwise, execute a poll on the discovery receiving socket with the lowest interval as timeout. The poll may return before this time if an advertisement is received or a new announcement is registered (see items 235, 236

and 243). If an advertisement has been received, read it from the socket recording the source address and port. Then parse the advertisement passing in the source address as a parameter (see item 244). If the advertisement can be successfully parsed, extract the EID of the node that sent the advertisement. Ignore advertisements sent from the node on which this BD is running (multicast will result in a send to the sending node which it may not be possible to filter out). Otherwise pass the remote EID and addressing information for the remote node to the *handle neighbor discovered* function (see item 239) which opens a new opportunistic link if necessary or reuses an exiting link previously created.

243. **Handle new announcement**. Notify the discovery agent thread that a new announcement has been registered. This is needed because otherwise the discovery agent thread might stay blocked in the socket poll for a long time before checking if it should send this new advertisement especially in the case where this is the first announcement registered when the poll timeout may be very large.

244. **Parse a received advertisement**. The format (defined by DTN2) with all items in network order is:
    - Type of IP convergence layer (One octet, TCP = 1, UDP =2)
    - Interval between advertisements (One octet unsigned integer, units of 100ms)
    - Total length of advertisement PDU (Two octets, unsigned integer)
    - IPv4 address of sending convergence layer interface (Four octets)
    - Port number of sending convergence layer interface (two octets, unsigned integer)
    - Length of sender EID (Two octets, unsigned integer)
    - ASCII character string of remote EID (not null terminated)

    Check the length of the data received matches the length given in the Total length field. If the IPv4 address given in the advertisement is 0.0.0.0, use the source address from the received packet as the IP address for the sending convergence layer. Combine the address and port into a string representation ("*<address>:<port>*"). Extract the remote EID. Return the convergence layer type, convergence layer next-hop IP address/port string, and remote EID.

245. **Shutdown IP Discovery Agent**. Set the shutdown flag to inform the discovery agent thread that it should terminate. Notify the thread as if a new announcement has been registered to ensure that the shutdown is actioned as soon as possible.

IP announcements require the following specialized functionality:

246. **Configure an *IP announcement* from the parameters passed in**. These are name of the convergence layer as a string (*tcp* or *udp* at present), the IPv4 address and port where it is possible to connect to the convergence layer and the interval between advertisements measured in seconds (see item 390). The address and port should have been configured as an interface so that this BD is ready to provided the specified convergence layer services at these addresses: For the UDP non-connection convergence layer this implies acceptance of incoming bundles at the address whereas for the TCP connection oriented convergence layer, the interface provides a passive TCP listener waiting for TCP connection requests. [It would probably be a

good idea to directly tie the announcement to an interface, so that announcements are only possible if there is a suitable interface. DTN2 does not do this.]

247. *Format an advertisement for the announcement.* The advertisement is described at item 244. The length of the datagram payload is rounded up to a multiple of 4 octets. [Why? This isn't checked and UDP datagrams have lengths in octets.]

4.9.3.4.2 Bonjour Discovery

The *Bonjour* (Apple™ Multi-cast DNS) technology can also be used to provide discovery agents for IP convergence layers. DTN2 implements a simple version of this which registers the TCP convergence layer running on the default port as a service. The Bonjour discovery runs as an independent thread which performs the registration and then browses for registrations by other nodes. In the event that another machine registers DTN convergence layer services, this node will be notified of the registration and will then execute the *handle neighbour discovered* functionality using the information received (see item 239).

4.9.3.4.3 Bluetooth Discovery

The *Bluetooth Discovery agent* can be used to provide discovery services for the Bluetooth RFCOMM convergence layer service (and in principle other services). The standard Bluetooth Inquiry mechanism is used to detect potential partners and retrieve their service offerings. Note that the Inquiry mechanism used by Bluetooth is very time consuming for the master node running the Inquiry. An Inquiry sequence takes at least 10.24 seconds and may well take longer. The Inquiry is run periodically from a separate thread to avoid blocking the bundle daemon core thread during the Inquiry. If this bundle agent node finds another node also offering the RFCOMM convergence layer service, retrieve the service parameters and execute the *handle neighbour discovered* functionality using the information received (see item 239).

4.9.4 Link Management

As described in Section 4.9.1.4, *links* model an actual or potential contact with a remote peer for sending and/or receiving of bundles. *Links* come in various flavours depending on the degree of persistence of the link and the way in which contacts are instantiated. Some functionality is common to all types of links but there is also some specific functionality for each type of link.

4.9.4.1 Link Functionality Common to all Link Types

The link records statistics relevant to the link. The following statistics are probably appropriate:

- Number of contact attempts for the link.
- Number of successful contacts (the number of successful openings).
- Number of bundles transmitted.
- Total number of octets transmitted.
- The number of bundles currently queued for sending and total number of octets in those bundles.
- The number of bundles currently inflight and total number of octets in those bundles. Inflight bundles are those that have been recently put on the wire. If

acknowledgements have been requested the inflight queue may contain more than one bundle. Bundles are deleted from the inflight queue when the bundle core processes the *Bundle Transmitted* event.

- Total number of bundles successfully cancelled. This is incremented when the bundle core processes the resulting *Bundle Send Cancelled* event.
- Link uptime. The length of time for which a contact has been open on this link in seconds. The start time for each contact is recorded in the contact data and the uptime is updated when the contact on the link is closed in the *Contact Down* event handler. If a contact is open when the statistics are read, the uptime for the current contact has to be added to the uptime recorded for previous contacts. For an always on contact that might the whole life of the link!
- Transmission throughput in bits per second. Calculated as the total number of octets transmitted converted to bits and divided by the uptime. Note that this may not be quite accurate if there is a bundle transmission in progress, particularly if it is a bundle with a large payload.

A *link* maintains state about the *convergence layer* that it is using (e.g., the transport address that the convergence layer is using for this particular address, and when the link is active, the thread(s) being used to handle asynchronous sending or receiving of data).

When a *link* has an active *contact* in progress, the link stores information about the contact in progress, and the convergence layer information may be expanded if necessary.

A *link* may also potentially store information relating to how it is used by the *router* that is active in the BD. For example, the Table Based Router model stores a list of *deferred bundles* that it wishes to forward on the link if the link *send queue* is full (see Section 4.8.2).

The following functionality is common to all links:

248. **Create a new link**. Provides a common entry point from which a link of a specified type can be created. The parameters to this function are:
    - A textual name for the link,
    - The type of link requested
    - The convergence layer to be used
    - The transport layer address for the remote peer (next-hop) as a string in a form usable by the specified convergence layer. The supplied string is parsed by the *parse next-hop* functionality provided by the relevant convergence layer (see items 312 and 329).
    - A list of parameters and requested values to be used by the link and associated convergence layer instance (see item 396).

    A *link* of the specified type is instantiated (see Section 4.9.4.2) passing the name, *convergence layer* and *next-hop* address to the creation routine. The parameter list is then passed to the instantiated link to pick up any link level parameters used to configure the link. Any remaining parameters are passed to the associated *convergence layer* to configure the specific convergence layer functionality. Create the bundle *send queue* and the bundles *inflight queue*. Record that the *send queue* and *inflight queue* are empty (no bundles queued and no octets queued in both queues). Modifications to these queues need to be protected by a lock as they are

performed from various different threads. By default links are created in state UNAVAILABLE but specific link types may implement a *set initial state* function that selects another state (e.g., an *always on* link will start in state OPEN).

249. **Delete a link**. Deleting a link is potentially quite complex and has to be managed in stages to ensure that no dangling pointers are left around. This functionality should not be called when there is an active contact associated with the link or the link has already been 'deleted'. The contact has to be closed before using this function. Note that to avoid race conditions the link should be marked as deleted but the associated structure should not be destroyed immediately. [In DTN2 links are one of the clever reference counted objects - a form of garbage collection. The actual link structure is destroyed automatically once its reference count drops to zero. The deletion of a link is normally started by posting a *Link Delete Request* event (see item 86) to the bundle core. This checks that the link is not already marked as deleted and calls the *delete link* functionality in the Contact Manager (see item 343). This in turn calls the *delete link* functionality in the associated convergence layer (see items 278 and 328) that delete the convergence layer state information stored in the link. A *Link Deleted* event is then posted to the bundle core and is also actioned by *routers* that need to know that links are deleted. In the case of the *external convergence layer* deleting a link, these events need to be reordered so that the convergence layer *delete link* functionality is not called until the *Link Deleted* event has been processed. Otherwise there may be a deadlock caused by nested events. [N4C does not need this complexity.]

250. **Reconfigure a link**. Provided the link has not been 'deleted', call the convergence layer specific functionality to reconfigure the convergence layer level link parameters using a passed in parameter list (see item 405). Not all convergence layers support reconfiguration of links or can only support it in specific states. [DTN2 allows the link level itself to be reconfigured by the external convergence layer code. This is not needed for N4C…. but there may be a rethink if the concept of auxiliary metadata connections is implemented for PRoPHET routing. The connection parameters would be obtained from the metadata link and might alter for different sessions.]

251. **Parse generic link configuration parameters**. Links have a large number of parameters. A configuration command is provided (see item 406) to set these parameters that are then used when *opportunistic links* are opened as a result of a *discovery advertisement* or incoming request to an *interface*. The settings remain in force until changed by a further configuration command and are used for any *opportunistic links* opened until the next configuration command is issued:

- Remote EID. The EID of the remote peer.
- Reliability flag. Boolean indicating if reliable data transfer is required, assuming it is possible.
- Next-hop transport address. String form of the convergence layer specific address for the next-hop to which bundles are sent on this link.
- Maximum Transmission Unit. The maximum size of bundles that can be sent on this link. If proactive fragmentation is allowed, bundles larger than this size will

need to be fragmented. Otherwise this is the maximum size of bundle that can be sent on the link.

- Minimum and maximum retry interval. Constraints on the time between attempts to open an *on demand* link that has failed or been closed because it was idle or an always on link that has failed. Neither can be zero. Configured in seconds and defaults vary between link types.
- Idle close time. Number of seconds of idle time before the link is closed. Zero means never close. This value must be zero for always on links. Defaults to 30 seconds for *on demand* links and zero for *always on*, *opportunistic,* and *scheduled* links.
- Potential downtime. Conservative estimate of the maximum amount of time that a link is expected to be 'down' after an unexpected closure during normal operation. Defaults to 30 seconds. May be used by some types of routing to decide if a bundle should be rerouted if a contact closes while trying to send a bundle and the contact is not restablished within this time.
- Previous hop header flag. Flag indicating that a previous hop header block can be added to bundles sent on this link. Defaults to false.
- Cost. Weight of this link. Potentially used to prioritize the use of alternative links to a destination. Defaults to 100. [DTN2 does not appear to use this parameter. The DTLSR routing algorithm maintains its own idea of cost separately.]
- Low and high water marks both in terms of bundles and octets to control the amount of data queued for transmission on this link. High limits are used to report if the queue is full. Low limits can be used by routers to indicate appropriate times to rescan the pending bundles list.

252. **Link state transition control**. Specifies allowable state transitions as follows:

Proposed New State	Transition Allowed from Old State(s)
UNAVAILABLE	Any state
AVAILABLE	UNAVAILABLE or OPEN
OPENING	AVAILABLE or UNAVAILABLE
OPEN	OPENING, BUSY or (for opportunistic links only) UNAVAILABLE
BUSY	OPEN
CLOSED	OPEN or BUSY

Note: BUSY is a pseudo-state determined by the volume of bundles and octets in the *send queue*. A link notionally transitions into the BUSY state if the volume of data exceeds the high water mark (see item 255) and returns to the OPEN state when the *send queue* drains to the point where there is less than the low water mark (see item 256).
CLOSED is a transient state. A link tat transitions to CLOSED will immediately and automatically transition onwards to either AVAILABLE or UNAVAILABLE (see item 84).

253. **Open a link**. Transition to state OPENING if allowed according to item 252. Create a new contact for this link and call the *open contact* for the associated convergence layer (see items 279, 313 and 329). Record the contact attempt in the link statistics.

254. **Close a link**. Check that the link has an open contact (states OPEN or BUSY). Log an error if not. Execute the close contact functionality for the associated convergence layer (see items 280, 310, 321 and 331) and check that the convergence layer information has been removed from the link information store. Delete any remaining information about the previous contact.

255. **Check if the link *send queue* is full**. Compare the number of bundles and octets queued with the corresponding configured high water marks (see item 251). If either limit has been exceeded the queue is full and the link is deemed to be in state BUSY.

256. **Check if the link has space to accept a new bundle for sending**. If both the number of bundles and the number of octets queued in the *send queue* are less than the corresponding configured low water marks (see item 251), the queue is deemed to have space and the link is in state OPEN.

257. **Add bundle to *send queue***. Lock the link *send* and *inflight queues* for the duration of the operations. Check the bundle is not already in the *send queue*. If so do not add the bundle and return an error. Otherwise add the bundle to the tail of the *send queue* and increment the count of bundles and octets queued accordingly.

258. **Delete bundle from *send queue***. Lock the link *send* and *inflight queues* for the duration of the operations. Delete the bundle from the *send queue* if it is in the queue. If not return an error. Otherwise decrement the count of send bundles queued and number of octets queued. Return success.

259. **Add bundle to *inflight queue***. Lock the link *send* and *inflight queues* for the duration of the operations. Check the bundle is not already in the *inflight queue*. If so do not add the bundle and return an error. Otherwise add the bundle to the tail of the *inflight queue* and increment the count of bundles and octets queued accordingly.

260. **Delete bundle from *inflight queue***. Lock the link *send* and *inflight queues* for the duration of the operations. Delete the bundle from the *inflight queue* if it is in the queue. If not return an error. Otherwise decrement the count of inflight bundles queued and number of octets queued. Return success.

261. **Output statistics**. Generate a textual dump of the current state of the link statistics. The statistics should be locked while this is performed.

4.9.4.2 Functionality for Specific Link Types

4.9.4.2.1 Always On Link Specific Functionality

The only specific functionality needed for always on links is:

262. **Set initial state**. An *always on link* should change to state OPEN immediately on creation assuming the creation is successful. This is done by posting a *Link State Change Request* event with the bundle core for a change to state OPEN. The event is

processed by the bundle core which results in a call to the bundle actions *open link* routine (see items 84 and 111).

4.9.4.2.2 On Demand Link Specific Functionality

The only specific functionality needed for *on demand links* is:

263. **On creation set the state to AVAILABLE and set a default 30 second link idle timeout**. See item 264.

264. **Set initial state**. Post a *Link Available* event with the bundle core to reflect the initial state change. The event is processed by the bundle core (see item 82). If the link is not marked as deleted (unlikely at this point!) the event is also processed by the contact manager and the installed router.

4.9.4.2.3 Opportunistic Link Specific Functionality

No specific functionality is needed for *opportunistic links* - they start in UNAVAILABLE state.

4.9.4.2.4 Scheduled Link Specific Functionality

Scheduled links maintain some extra information which is the *future contact* set. Each entry contains an absolute *start time* and a *duration*. The link starts in state UNAVAILABLE. The following extra functionality is needed:

265. **Add a future contact**. Insert the indicated contact into the set.

266. **Delete a future contact**. Delete the indicated contact from the set.

267. **Convert a future contact into an active contact**. [Not implemented in DTN2].

268. **List current future contact set**.

4.9.5 Contacts

As described in Section 4.9.1.5, a contact models an active link that is in communication with a remote peer through the BD at a next-hop node. The contact maintains some state information about the contact:

- The time when this contact started.
- The duration of this contact in seconds. This is zero if the duration is not known in advance. This is relevant for scheduled links where the duration of the contact is set by the router. Also the duration of the contacts associated with a link are summed up into the *link uptime* statistic for the link (see item 77 and Section 4.9.4.1).
- The link with which the contact is associated
- State information stored by the convergence layer relating to this link and contact.
- An approximation to the bandwidth of the link in bits per second.
- An estimate of the latency of the link in milliseconds(?).

[The last two items are not used by any type of contact currently implemented in DTN2.]

A *contact* is a convenient container for the various pieces of state information about the contact itself and the additional information required by the *convergence layer* when managing an active communication session, but does not require any other specific functionality..

4.9.6 Convergence Layer Abstraction

All *convergence layers*, whether connection oriented or non-connection oriented implement the services defined at a very high level in Section 7 of [RFC5050]. As explained in Section 4.9.1.1, the BD manages communications with BDs in other nodes through the abstractions of *contacts*, *links* and *interfaces*. The *Convergence Layer* model encapsulates the protocol used for communication on a particular link during a contact, and manages the interactions between the local communication devices and the BD in establishing a link, closing down a link, and handling date reception and transmission on the link.

Note that it is assumed that *convergence layers* will not be dynamically defined. In a production system it is extremely unlikely that one would want to add a new convergence layer dynamically although it would be nice if they could be added via plug-ins. [DTN2 provides the external convergence layer mechanism but this uses XML encoding for communication and is probably rather slow for production use.]

When the BD starts up the various internal convergence layers are instantiated and initialized. The External Convergence Layer can be started via a management command if it is available:

269. **A list of *convergence layer* instances is created and filled with single instances of each type of convergence layer available in the BD**. Depending on the platform and compilation time options the following convergence layers may be available - the string in brackets is the name of the type of convergence layer used when associating the convergence layer with a link, interface or discovery announcement:

    - Null Convergence layer (*null*).
    - Serial Link Convergence layer (*serial*).
    - TCP Convergence layer (*tcp*).
    - UDP Convergence layer (*udp*).
    - Ethernet Frame Convergence Layer (*eth*) - Linux platforms only.
    - Bluetooth RFCOMM Convergence Layer (*bt*) - if compiled into BD.
    - NORM Convergence layer (*norm*) - if compiled into BD.
    - File system Convergence Layer (*file*) - code is incomplete in DTN2 and is not included.
    - External Convergence Layer (*ecl*) - if compiled into BD and then started through configuration interface.

All *convergence layers* have to provide the following functions although the details of what has to be performed will be specific to the convergence layer because it will depend on the characteristics of the communications end point and the protocol to be used on the resulting links:

270. **Bring a new *interface* up**. This involves instantiating the communications end point using addressing and other relevant parameters as required by the platform operating system and protocols (see item 393). The end point is than attached to the convergence layer and a receiver thread is created so that incoming data can be accepted on the interface if appropriate. The format and semantics of the data depend on the convergence layer. Depending on the type of convergence layer, the interface may also *listen* for incoming connections that are intended to result in the instantiation of an opportunistic link.

271. **Take down an existing *interface***. Destroy the listening thread and close down the associated communications end point.

272. **Open a *contact* associated with a *link***. This function creates a new *contact* that uses the parameters configured for the *link* and, if necessary (depending on the *convergence layer* type), creates threads to manage outgoing data and incoming data on the link. On successful completion a *Contact Up* event is posted (see item 76).

273. **Close down an open *contact***. This function provides clean up of the connection state when a *link* needs to close a connection. It does not post a *Contact Down* event but is called in response to some other part of the system generating this event.

274. **Notify a *link/contact* that a bundle has been queued for transmission**. Depending on the *connection layer* type, this can either trigger the sending thread to deal with the queued bundle, or in less sophisticated cases can actually handle the sending of the bundle.

275. **Request cancellation of transmission of a bundle previously queued by the bundle daemon if it hasn't yet been transmitted**.

276. **Provide any operations needed to shutdown the convergence layer**.

4.9.6.1 Connection Oriented Convergence Layer Model

All *convergence layers* that maintain a *connection* (e.g., a TCP connection) to *next-hop peers* use this model. As such, a *convergence layer* conforming to this model manages all communication to/from the main bundle daemon thread, handles a main thread of control for each connection, and dispatches to the specific *convergence layer* implementation to handle the actual wire protocol.

The connection operates asynchronously controlled by a *connection message queue* into which commands are pushed in response to commands sent from the bundle core. The commands are SEND_BUNDLE, CANCEL_BUNDLE, and BREAK_CONTACT. When in an idle state, the connection thread blocks on this queue as well as the socket or other connection object so it can be notified of events coming from either the daemon or the peer node.

When the *contact* comes up, a new connection is created including the main thread of control for the connection. In this connection oriented model, the contact may have been initiated from this node or the remote node. Using the terminology normally used for TCP sockets, the local end of the connection is either described as *active* when it initiates the contact or *passive* when the connection is initiated from the remote end and a previously

configured local interface is used to handle the passive end of the protocol. The connection main thread is instructed to perform '*connect style*' (for local initiation) or '*accept style*' functionality depending on which end of the connection initiates the connection. For some types of convergence layer, the *accept style* is not used. For example with serial links, both ends use pre-configured interfaces.

To enable backpressure, each connection has a maximum queue depth for bundles that have been pushed onto the link *send queue* but have not yet been sent or registered as *in-flight* by the CL. The state of the link is set to BUSY when this limit is reached, but is re-set to AVAILABLE or OPEN as appropriate when the queue has drained (it would be set to AVAILABLE if the contact had ended when the queue drained). [Note that in DTN2 this is not explicitly implemented in the link state machine, but by requiring that routers check the queue state explicitly before trying to queue a bundle. Not all of them do.] By default, there is no hard limit on the number of bundles that can be *in-flight*, instead the limit is determined by the capacity of the underlying link. Generally only the bundle at the tail of the *in-flight queue* is actually being transmitted. Others in the queue may be waiting on acknowledgements.

The hardest case to handle is how to close a *contact*, as there is a race condition between the underlying connection breaking and the higher layers determining that the link should be closed. If the underlying link breaks as detected by a timeout or, in the case of an *on demand* link, goes idle for an extended period, a *Contact Down* event is posted and the thread terminates. In response to this event, the daemon will call the *close contact* functionality (see item 280). In this case, the connection main thread has already terminated so it is cleaned up when the *contact* container is destroyed.

If the *link* is closed by the daemon thread in response to a management command or a scheduled link's open time ending, then the *close contact* functionality will be called while the connection is still open. The *main connection thread* is informed by sending it a BREAK_CONTACT command. Reception of this command closes the connection and terminates the thread when it is done. All this logic is handled by the *break contact* functionality in the specific connection class (see items 310 and 321).

Finally, for bidirectional protocols, *opportunistic links* can be created in response to new connection requests arriving from a peer node.

The following functions are provided in the BD thread:

277. **Initialise a *link* associated with a *connection oriented convergence layer* and set parameters**. The main parameters that are needed are a flag to determine if reactive fragmentation is enabled, the timeout for data reception, and buffer lengths for the send and receive buffers (see item 396 for a listing of all possible parameters). The parameters are stored in association with the *link* since the *convergence layer* is a separate single entity.

278. **Delete a *link* associated with a *connection oriented convergence layer***. Delete the link associated parameters from the link information.

279. **Specific support for opening a *contact* using a *connection oriented convergence layer***. This involves creating the main thread that handles commands and data events for this connection.

280. **Specific support for closing the *contact* when using a *connection oriented convergence layer***. This involves stopping the subsidiary thread by queuing a BREAK_CONTACT message for the connection thread.. Once the thread has stopped it may be necessary to clean up a partially transmitted bundle and/or a partially received bundle. If *reactive fragmentation* is not enabled or the current inflight bundle had not started transmission or, for a reliable link, no acknowledgements had been received for the data transmitted, the current inflight bundle needs to be pushed back onto the link *inflight queue* as a complete bundle [I am unsure about this. I would have thought it ought to be pushed onto the front of the *send queue* but that is not what DTN2 does - the *inflight queue* is emptied shortly thereafter], and otherwise a *Bundle Transmitted* event is posted on the bundle core indicating the amount of the bundle that has been acknowledged. The *inflight queue* is then emptied. If *reactive fragmentation* is enabled and a partial bundle has been received, a *Bundle Received* event with the partial length should be posted for the bundle core. Thereafter the *incoming* bundle queue should be drained and the connection thread *command message queue* drained before destroying the connection.

281. **Specific support for the *bundle queued* function in the *convergence layer* model**: A BUNDLES_QUEUED message is queued for the main connection thread to trigger the start of transmission in case the connection is idle.

282. **Specific support for the *cancel bundle* function in the *convergence layer* model**: A CANCEL_BUNDLE message is queued for the connection thread to request cancellation of the bundle. If the contact has been closed down before this function gets called, a *Bundle Send Cancelled* event should be posted on the bundle core, as the cancel has effectively happened.

The main connection thread for each *contact* expects to be supported by specific functionality for each type of convergence layer. The main connection threads provide the following functionality:

283. **Instantiating a new *main connection thread***. This creates the thread but doesn't run it. In particular it creates the *connection message queue* for communication from the main BD thread. It also performs any *convergence layer* related instantiation. The instantiation records whether this is the *active* or *passive* end of the connection, the time to block for on each iteration of the run loop while waiting for any events, and sets any optional parameters relevant to the specific convergence layer.

284. **Provide a main loop for the *main connection thread***. This loop monitors two places where events may be generated: the contact connection end point for both reading and writing of data from the remote end point and the *connection message queue* for commands from the BD thread. Depending on whether the instantiation is active or passive the *connect* or *accept* function is called. Whichever function is called, if the *contact* is successfully initiated, a *Contact Up* event (see item 76) is

posted on the bundle core. This function will be specific to the *convergence layer* being instantiated. The thread then enters a continuous loop that is only exited when the contact is broken, either as a result of the communication link failing, an idle timeout in the case of *on demand* links or explicit command from the BD. In the loop, the thread blocks for the timeout configured at item 283 waiting for activity on the communication link or the message queue. Depending on the activity detected a convergence layer specific function is called to handle the incoming message from the BD (*handle bundles queued*, *handle cancel_bundle* or *break contact*), availability of write buffer space when there is more data to send (*send pending data*) or other activity on the communications endpoint (*handle_poll_activity*). If the block ends due to the timeout without any activity being detected, the function *handle poll timeout* is called.

285. **If the contact ends because of explicit action in the local BD** (i.e., user action or idle timeout), the convergence layer specific *disconnect* function is called to close down the communication end point. This is not necessary if the link has broken due to communications failure. If the contact ends for any reason except explicit user action, a *Link State Change Request* event is sent to the BD to ensure the OPEN link is closed (see item 84).

286. **When a new contact is initiated**, it may be either as a result of 'top down' action coming from the BD or 'bottom up' action resulting from an accepted low level connection that receives details of a proposed connection from the remote end; this constitutes the initiation of an opportunistic link contact.. In the top down case, the contact information has already been created and associated with a link before the information is received from the remote end. In the bottom up case, there will generally not be a link associated with the new contact. In this case a new *opportunistic link* must be created by the contact manager and associated with the contact (see item 350).

4.9.6.2 Stream Oriented Convergence Layer Model

Many of the *connection oriented convergence layers* use reliable, in-order delivery protocols (e.g., TCP, SCTP, and Bluetooth RFCOMM). The goal of this model is to express and share as much functionality as possible between protocols that have in-order, reliable, delivery semantics. The model implements the protocol defined in the TCP Convergence Layer specification [TCPclayer], generalized to allow it to be used on any transport that provides similar protocol semantics to TCP.

For the protocol, bundles are broken up into configurable-sized segments that are sent sequentially. Only a single bundle is inflight on the wire at one time (i.e., segments from different bundles are not interleaved on the wire). When segment acknowledgements are enabled (the default behaviour), the receiving node sends an acknowledgement for each segment of the bundle that was received. Since the transport protocols guarantee in-order delivery and a single channel is used in each direction for transmission of data, progress on both sending of a bundle and receiving of acknowledgements can be recorded by a single counter for each of them. [DTN2 uses a sparse bit map which seems to be overkill given the transport semantics.]

A queue is maintained for each of *inflight* and *incoming bundles*. This is necessary to allow sending and receiving of bundles to be overlapped with receiving and sending of corresponding acknowledgements. In each case, the bundle at the tail of the queue is having its data sent or received as appropriate, whereas the bundle at the head of the queue is waiting for or sending acknowledgements. In normal operation we would expect to have zero, one or two bundles in each queue, but in exceptional circumstances acknowledgements could get significantly behind sending of data and additional bundles could be waiting to send or receive acknowledgements.

Outgoing bundles are managed by the *inflight bundle status* and incoming bundles by the *incoming bundle status.*

The *inflight bundle status* is used to record state about bundle transmissions. To record the segments that have been sent, we fill in the *last transmitted index* as we send segments out until the whole bundle has been sent. If there is a new bundle awaiting transmission in the *send queue*, it is moved to the *inflight* queue and transmission of the new bundle commences. If (positive) acknowledgements are being sent, and received (the semantics requires that both directions of transmission must do acknowledgements or neither does), the bundle remains in the *inflight* queue. Otherwise, the bundle core is notified that the bundle has been transmitted and the bundle is deleted from the *inflight* queue. As acknowledgments arrive, we extend the *acknowledged transmission data index* according to the data in the message. As indicated above, the acknowledgements apply to the bundle at the head of the *inflight* queue, which may or may not be the same bundle that is being transmitted at the time the acknowledgement arrives. Once the whole bundle is acknowledged, the entry is removed from the *inflight bundles* queue, the bundle core is informed of a successful transmission and subsequent acknowledgements are applied to the new bundle at the head of the *inflight queue*..

The *incoming bundle status* is used to record state about bundle reception. The incoming data goes to construct a new bundle that is referenced at the tail of the *incoming bundles queue*. The *received data index* is updated with the amount of data that has been received, including partially received segments. The expected size of the incoming bundle is retrieved from the primary block of the incoming bundle and when the received data index indicates that the whole bundle has been received, the bundle received is notified to the bundle core, and reception of a new bundle started if more data arrives. If acknowledgements are being sent, they will apply to the bundle at the head of the *incoming bundles queue*; as with the transmission side, this may or may not be the same bundle for which data is being received. A record is kept of the last *acknowledged received data index*. When this reaches the size of the bundle being acknowledged, the bundle is removed from the *incoming bundles queue* and acknowledgement of the next bundle in the queue, if any, commences.

Keepalive messages are sent back and forth to ensure that the connection remains open if there are no bundles to be sent and the connection is configured to use *keepalives*. In the case of *on demand links*, a configurable *idle timer* is maintained to close the link when no bundle traffic has been sent or received. Links that are expected to be open but have broken due to underlying network conditions (i.e., *always on* and *on demand links*) are reopened by a timer that is managed by the *contact manager*.

Flow control is managed through the poll callback functions that are invoked from the run loop of the connection oriented model main connection thread. In *send pending data* (see item 291), we check if there are any acknowledgments that need to be sent, then check if there are bundle segments to be sent (i.e., acknowledgments are given priority). The only exception to this is that the connection might have write blocked in the middle of sending a data segment. In that case, we must first finish transmitting the current segment before sending any other acknowledgments (or the shutdown message), otherwise those messages will be consumed as part of the payload.

To make sure that we don't deadlock with the remote end point, we always drain any data that is ready on the channel. All incoming data and acknowledgement messages record state in the appropriate data structures, then rely on *send pending data* to send the appropriate responses.

This model also handles the contact establishment protocol and data exchange protocol described in Sections 4 and 5 of the DTN TCP Convergence Layer Protocol [TCPclayer]. Although the standardization process currently specifies this protocol only for the TCP protocol, the framing and operation are appropriate for any protocol that follows the stream paradigm such as SCTP or Bluetooth RFCOMM. [DTN2 has an implementation for Bluetooth RFCOMM using the stream, model and this protocol. SCTP has not yet been implemented - there are clearly some issues to be worked out about the interaction of multiple paths with DTN forwarding.]

The following functions are provided in the BD thread:

287. **Initialize the link and set parameters**. This includes selection of the acknowledgements method (positive or positive/negative), interval between *keepalive messages* when the link is idle and the length of segments to be sent. Notify the link layer that the connection is reliable or otherwise.

The majority of the functionality in this model is associated with the main connection thread that handles communication for this thread. This thread is created and managed in the *Connection Oriented Convergence layer* model (see Section 4.9.6.1) and provides some generic stream-related parts of the specific convergence layer functionality described in items 279 to 285. The following functionality is provided for the *main connection thread*:

288. **Initiating the contact protocol**. Create a *send buffer* for assembling the outgoing data and a *received buffer* for receiving incoming data. When the low level connection is established, a *contact header* (see Section 4 of [TCPclayer]) is constructed and sent to the remote endpoint (this is done whether the current BD is the initiator or acceptor of the connection) and the timers for the sending of keepalives and checking for an idle link are initiated if required. The contact header sent indicates whether any of acknowledgements, reactive fragmentation and bundle refusal (negative acknowledgements) are to be supported for this contact. [Note DTN2 does not support bundle refusal.] The connection thread then waits to receive the *contact header* from the communications partner.

289. **Checking a received *contact header***. This must be the first message from the remote partner. Ensure that the received *contact header* conforms to the syntax as specified in Section 4 of [TCPclayer]. If not correct, terminate the contact using the

break contact procedure (see item 285). If correct, record the EID asserted by the remote partner and check that there is a contact registered for this EID. There will be a contact in existence if the connection has been initiated from this node, but if the *passive* interface has accepted an incoming TCP connection or similar function for other transport protocols, the connection may represent a new *opportunistic link* and the contact will not yet have been created. The functionality at item 286 is invoked to setup a new *opportunistic link* and associate it with the contact. Complete the negotiation of support for any of acknowledgements, reactive fragmentation and bundle refusal. [DTN2 does not support bundle refusal.] Start the *keepalive timer*, if requested, using the negotiated interval. On successful completion inform the bundle core that the contact is up by posting a *Contact Up* event (see item 76).

290. **Handle bundles queued**. No special action is needed for the stream convergence layer as the connection thread is regularly monitoring the queue.

291. **Send pending data**. This function is called to check if there is space in the *send buffer* for the communication channel, and, if there is, use it to send, in order of priority, the remainder of any partially completed segment send that had to be terminated because of lack of buffer space, any acknowledgements for data that has been received (if segment acknowledgement is enabled) and data from any queued bundles awaiting transmission, continuing with the next segment of a partially transmitted bundle if there is one, or starting a new bundle if there are any queued.

292. **Start transmission of a new bundle**. Move the bundle from the link's *send queue* to the *inflight queue* for the link (see Section 4.9.4 and using items 258 and 259), record this bundle as the one currently inflight, and initialise records of the sent and acknowledged segments of the bundle. Note that the records must be at the octet level, since buffer constraints in the underlying protocol may prevent whole segments being transferred to the transport protocol in one call, and acknowledgements may arrive for any number of octets. The *inflight queue* needs to be a queue since even though only one bundle is inflight at a time, the receiving of final acknowledgements for one bundle may be overlapped with sending the start of the next bundle. [DTN2 uses a sparse bit map to maintain these records. Since both the sending and acknowledgement work on a high water mark mechanism, i.e., the sending sends the octets in sequence after the last ones sent and acknowledgement in [TCPclayer] supplies the offset of the highest contiguous data received, the use of a bit map seems rather overkill here. The same is true for the receiving side. Since we are dealing with a stream at this layer any out of order delivery is hidden by the underlying transport protocol.]

293. **Send next segment**. Determine if there is any more of the current inflight bundle to be sent. If so, assemble the next segment into the *send buffer* prefixed with the correct header (see Section 5.2 of [TCPclayer]). The next segment's worth of bundle data is obtained using the *produce* routine for the bundle (see item 151). If all the bundle has been sent invoke the *finish bundle transmission* functionality (see item 295). Otherwise invoke item 294 to actually transmit the data. [Note that DTN2 uses a resizable linear buffer to handle the intermediate buffering between the bundle producer and the transport protocol end point. This probably not the best choice. A

ring buffer would probably be a better choice. As it stands it appears that the bundle producer may be invoked multiple times if the transport protocol write blocks.]

294. **Send more data**. Invoke the transport specific send routine to send as much data as possible from the send buffer (e.g., see item 323). If the transport protocol blocks because the write buffer is full this may leave data still to be sent when possible. Record what has been sent. [Note that DTN2 potentially records some data as having been sent when it has not actually been passed to the transport protocol but is only in the transmit buffer. This is probably only a technical issue as the data will be sent before we get an acknowledgement for it!] If the bundle has now been completely sent, record that there is no bundle currently inflight (ready to start a new ones, if there are any waiting) and execute the *check completed* functionality (item 296).

295. **Finish bundle transmission**. Record that no bundle is inflight at present and execute the *check completed* functionality (item 296).

296. **Check if a bundle has been sent and acknowledged completely**. Check that the bundle has been sent completely (indicated by the currently inflight bundle is not the one for which this check is being performed). If so, and if acknowledgements are being sent, verify if acknowledgements have been received for the whole bundle. If not wait for more acknowledgments to arrive. Otherwise send a *Bundle Transmitted* event to the bundle core (see item 57).

297. **Send a *keepalive message***. This functionality will be called when the *keepalive timer* expires if keepalive functionality is configured for this connection. If data is being sent or there is some pending data waiting to be sent when this happens suppress the keepalive (the data being sent serves the same function). Otherwise push a *keepalive message* as specified in Section 5.5 of [TCPclayer] into the *send buffer* and invoke the specific transport protocol to send the data (e.g., see item 323). Restart the *keepalive timer*.

298. **Cancelling a bundle scheduled for transmission**. This can only be done if the bundle has not yet started being sent. If any or all of the bundle has already been sent, then it is not possible to cancel it and this routine does nothing. Otherwise delete the bundle form the *inflight queue* (see item 260) and post a *Bundle Send Cancelled* event for the bundle daemon (see item 62).

299. **Manage inactivity timeouts**. These will be caused when no data is sent or received on the lower level transport connection for a period of time. If *keepalives* are configured for the connection and no data or *keepalive message* has been received for the configured multiple of the keepalive time, then break the connection and shutdown the specific protocol connection. If an *idle timeout* is configured for this link and the inactivity period is longer than the idle timeout, then break the contact and shutdown the specific protocol connection. Determine if a *keepalive message* should be sent.

300. **Send pending acknowledgements**. Determine if any data has been received which extends the contiguous data block received in the bundle [With a stream protocol, it doesn't appear that it could be otherwise than contiguous. DTN2 has elaborate means to deal with non-contiguous receives using sparse bit maps. This

appear to be spurious because data received is necessarily contiguous, and the convergence layer specification doesn't allow otherwise because it only gives the segment length and not a position.] If so and segment acknowledgements are configured on, generate an Acknowledge segment as specified in Section 5.3 of [TCPclayer] and put it into the *send buffer* if there is space. If there is no space just do nothing and wait till next time round. Send any generated acknowledgement using the specific convergence layer (e.g., see item 323).

301. **Check if a *keepalive message* needs to be sent**. Check if longer than the *keepalive interval* has elapsed since any data or the last *keepalive message* was sent. If so use item 297 to send a *keepalive message* unless data sending is in progress but not yet recorded..

302. **Process received data**. This takes the raw data received by the specific *convergence layer* and splits it into records as described in Sections 4 and 5 of [TCPclayer]. If the contact has not yet been verified as 'up' , the received data should be a *Contact Header*. Despatch the received data for processing as at item 289. Otherwise the received data should be (part of) a data segment, an acknowledgment signal (if acknowledgements are enabled), a bundle refusal signal, a keepalive signal or a shutdown signal. In all cases record the time when data was received in order to manage *idle timeouts* if required. If a data segment is in progress, pass the data to the segment data handler for processing (item 304). If there is remaining data after successful processing of the data segment or no incoming data segment was expected, examine the first octet received to determine the type of the next segment and the associated flags. According to the type dispatch the remaining data to be processed as the start of a new data segment (item 303), an incoming acknowledgement (item 305), a bundle refusal (item 306), an incoming keepalive (item 307) or a shutdown request (item 308). If the first octet does not contain one of the appropriate codes there has been a protocol error and the contact should be broken. Note that it may be necessary to wait for additional octets to be received by the specific convergence layer if not enough data has been received to process the segment header for each type of segment. In that case the received data is left in place pending receipt of more. This may never happen if the contact has broken - the poll timeout will deal with cleanup (see item 299).

303. **Start receive of new incoming data segment**. Verify that the initial segment in the *received buffer* has the start (**S**) flag set in the segment header. If not report a protocol error and break the connection. If so, add a new incoming bundle to the tail of the list of incoming bundles (some previous bundle(s) may be awaiting the sending of acknowledgements. All incoming data is added to the last bundle on the list. Decode the SDNV giving the length of the segment and record that we are expecting this much data. If the end (**E**) flag is set, the expected total length of the bundle can be calculated and recorded. Set the records of received and acknowledged data for this bundle to zero ready for processing the bundle data. Use item 304 to process any data already received.

304. **Handle incoming data segment data.** As noted in item 303, the incoming data in the *received buffer* should be added to the last bundle on the incoming list. Pass any received data obtained by the specific convergence layer, up to the amount required

to complete the current segment as recorded in item 303, to the *consume* routine of the bundle which will assemble and ultimately decode the bundle from the raw received data (see item 147). Record the actual amount of received data transferred for the incoming bundle and decrement the amount of data expected in the segment by the same amount. Delete this data from the received data buffer. If this amount of data completes the segment, check if the bundle is completed by executing item 309.

305. **Handle bundle acknowledgement**. Break contact if bundle acknowledgments are not expected for this contact. Decode the SDNV giving the acknowledged length of the data if enough octets have been received. Verify that there is an inflight bundle to which this applies (the one at the head of the queue of *inflight bundles,* which may possibly have been completely sent by now). If the acknowledged length has decreased since the previous acknowledgements, report a protocol error and break contact. Otherwise record the amount of data acknowledged so far. If the acknowledged length is equal to the total length of the completed bundle, this bundle has now been successfully transmitted. Record this and post a *Bundle Transmitted* event with the bundle core (see item 57). Note that the *Bundle Transmitted* event is only posted from here if the contact was requesting acknowledgements. If the acknowledged length is longer than the recorded bundle total length report a protocol error and break contact. [Note DTN2 does not do this.]

306. **Handle incoming bundle refusal**. [Not implemented by DTN2. DTN2 just breaks contact as it should have been signalled as not supported and peer should not have sent it. The semantics of this capability are very complex. In particular it is unclear from the specification whether *reactive fragmentation* should be applied to a bundle that has been refused. It strikes me that it would be better to modify the TCP convergence layer specification to signal the total length of the bundle in the first segment header.]

307. **Handle incoming keepalive**. If keepalive signalling was not enabled this is a protocol error and we should break contact. [DTN2 does not check - the keepalive can just be ignored but should be logged as unexpected.] Record that the *keepalive message* has been received to show that the channel is still functioning.

308. **Handle incoming shutdown signal**. Determine if the shutdown segment includes the optional reason code and/or reconnection delay fields. If so decode the optional fields. The reconnection delay is an SDNV. [DTN2 treats it as a fixed width 2 octet unsigned integer which is inconsistent with [TCPclayer]. DTN2 also does nothing with the reconnect delay.] Invoke the break contact functionality (item 310).

309. **Check for completed incoming bundle**. If the segment with the bundle end (**E**) flag has been received and its data completely inserted into the incoming bundle, then check that the received length matches with the length of the bundle as calculated by the bundle constructor (see item 147). If not signal a protocol error and break the contact. If so, post a *Bundle Received* event with the bundle core (see item 56).

310. **Break contact**. A reason code will be supplied on each call. If contact breaking has already been requested return immediately (this prevents an infinite loop during shutdown: handshaking with the peer, and also several places in the code, may detect the need to shutdown.) Record that we are breaking contact. Determine whether a shutdown message should be sent to the contact peer. It should be sent if the break was requested by the user, if the link has been idle for too long, if there is version mismatch in the convergence layers or the peer requested a shutdown (handshaking). Otherwise no shutdown message is sent. The shutdown message will also be suppressed if we are in the middle of sending a data segment, as the peer would not recognize it. If a shutdown message is to be sent construct the shutdown message as specified in Section 6.1 of [TCPclayer] and add a reason code if appropriate. [We should add a reconnect delay if required. DTN2 never does this. I guess this would be a user configurable parameter. Note that [TCPclayer]allows for an 'infinite' reconnection delay. This could be interpreted as 'destroy the link' so it is never used again.] Call the *break connection* functionality in the connection convergence model (item 280).

4.9.7 Example Concrete Convergence Layers

4.9.7.1 TCP Convergence Layer

The TCP (Transmission Control Protocol) Convergence layer [TCPclayer] is an example of a Connection Oriented Convergence layer (see Section 4.9.1.2.2) and implements the Stream Convergence Model (see Section 4.9.6.2) for a TCP over IP connection through a socket communication endpoint. By default it listens on port 4556 for incoming connections, but this can be altered by configuration. The convergence layer has a version number (currently 3) that is used to ensure compatibility between BDs implementing this convergence layer. The local address for the link end point can be configured. In line with the Connection Convergence model (see Section 4.9.6.1), the TCP Convergence layer has functionality that operates on three separate threads: the main bundle core thread, and two specific threads that are used to listen for incoming connections (the Listener thread) and the Connection thread that manages reading and writing on a connected socket associated with a current Contact on a Link. The following functionality is implemented for the main bundle core thread:

311. **Parsing the link parameters**. Extract settings specific to the TCP layer from textual settings supplied by the user/management system. This is an IP address for the local side of the connection where the Listener will listen. [DTN2 only supports IPv4 addresses. It should support IPv6 also.] Any remaining parameters are passed to the generic Stream Model for further parsing. See item 396 for a listing of all the possible parameters.

312. **Parse next hop address**. Extract the remote peer IP address and port from information accessible in the *link* data. Use the default port if not explicitly specified. Verify that an explicit address has been given rather than the 'any address' constant (INADDR_ANY in IP parlance). It may be possible to look up a textual hostname in the local hostname database (but of course, do not expect DNS to work!)

313. **New connection**. Attaches the TCP specific functionality for a new connection to the *main connection thread* created by the Connection Oriented model (items 279 and 283) and links it to the Stream model functionality (item 287).

314. **Handle creation of a new listening interface (*interface up*)**. When instructed by the *interface manager*, create a passive listener TCP port and bind it to the specified local IP address and port. Create and start a Listener thread to monitor incoming connection requests on the interface and record the Listener with the interface (item 227).

315. **Handle destruction of a listening interface (*interface down*)**. Stop and delete the listener thread under the control of the *interface manager* (see item 229). Note that this does not affect any existing connections that have already been opened, but will prevent future connections from being opened. *Links* that purported to use this interface need to be terminated when the contact next ceases. [DTN2 does not do this.]

Functionality for the listener thread consists of:

316. **Processing for a new accepted connection request from a remote address**. A new *TCP convergence layer connection* thread is created and started to handle the new *contact* (see items 317.2 and 320). The cloned socket supplied by the operating system **accept** call is passed in with the connection request .

Specific *TCP Convergence Layer* functionality for a TCP connection provided on the connection management thread:

317. **Create a new connection**. Two cases need to be handled:

317.1 **Create a new connection in response to a local request from the user or management interface (see item 396)**. Record the remote address and port (the *next-hop*) supplied with the request for use when connecting to the remote peer. A new TCP (client or active listener) socket is created and bound to the local address specified for this connection (if any) and set to be non-blocking so that reads and writes return immediately when actions would block waiting for operating system resources. The *Connection Convergence Layer* in which this is embedded is requested to initiate a connection via the new socket to the remote peer (see items 283 and 284) which invokes the *connect* functionality using the stored remote address and port.

317.2 **Create a new connection in response to a connection request arriving at the passive listening socket managed by the Listener thread**. Record the remote address and port (the *next hop*) obtained from operating system using information associated with the socket created by the incoming TCP connection request. The socket is set to be non-blocking. The *Connection Convergence Layer* in which this is embedded is requested to accept the connection via the socket to the remote peer (see items 283 and 284) which invokes the ‘accept’ functionality.

318. **Provide functionality to initialise the list of sockets that need to be 'polled' to detect activity for this connection**. This is just the single socket that results from the creation of the connection (see item 317).

319. **Perform a 'connect' for an active mode TCP socket**. Use the remote address and port previously recorded to perform the connect. If the connect succeeds immediately start the procedure to initiate the contact provided by the Stream layer (see item 288). If the connection fails break the contact (see item 310). Otherwise the connection will be notified as 'in progress'. In this case record an interest in the polling event that will occur when the connection eventually completes (or fails) and continue.

320. **Perform a 'connect' for a passive mode TCP socket**. In the case where the connection request is originated by the remote peer, the low level accept functionality has already been called in the Listener thread in order to retrieve the cloned socket that represents the connection and the TCP connection is already in existence. This function just needs to start the procedure to initiate the contact provided by the Stream layer (see item 288).

321. **Disconnect the connection**. At the TCP layer all that is necessary is to close the communication socket used by the *Connection thread*.

322. **Handle poll activity**. Examine any events recorded for the socket used by this connection. If a hang up or error event occurs break the contact (see item 310). If write readiness is notified this may either indicate that a previous connect has completed or the operating system transmit buffer has drained and we can send any pending data. If a previous connect has been recorded and it has succeeded, start the procedure to initiate the contact provided by the Stream layer (see item 288). If the previous connect failed then break the contact. Otherwise send any pending data (see item 323). If read readiness is notified read the data from the socket (see item 324) and then process it as specified in the Stream Convergence model (see item 302). Finally check if it is time to send a keepalive signal to the remote peer (see item 301).

323. **Send any pending data if possible**. If there is data in the *send buffer* (see item 288) write as much of it as possible it to the connection socket. Record the actual amount sent and delete it from the *send buffer*. If more data remains to be sent, set the poll events to monitor for the socket to become writeable again. If there is an error when writing break the contact (see item 310).

324. **Receive any pending data**. Read as much data from the connection socket as is available and will fit into the available buffer space in the *receive buffer* (see item 288). If there is an error when writing break the contact (see item 310).

4.9.7.2 UDP Convergence Layer

The *UDP (User Datagram Protocol) Convergence Layer* is a (simple) example of a *Non-connection Convergence layer* (see Section 4.9.1.2.1). It uses UDP over IP to transport bundles. It is generally not recommended for general purpose use, but can be used in conjunction with other protocols such as the Licklider Transmission Protocol [RFC5325] to

provide a carrier on an IP network. A draft specification for the UDP convergence layer exists as [UDPclayer].

The UDP Convergence Layer is significantly simpler than the TCP Convergence Layer (see Section 4.9.7) but offers no reliability and cannot guarantee in order delivery of data. Because of this it expects a bundle to fit into a single datagram (maximum length 65507).

By default the UDP convergence layer receives and sends data at (UDP) port 4556, but the local port is configurable.

A link/contact can be created in the local BD directed towards some specified node by means of a remote IP address and port number but this does not involve any exchange of bundles or setup packets between the nodes. Creating a contact does not guarantee that the nodes will be able to communicate over UDP convergence layer. You will only find out if the nodes can communicate when a bundle is sent.

A receiver thread is created when the interface using the UDP Convergence layer is opened. Depending on whether the creator of the interface specifies a remote IP address, this thread will immediately be able to receive UDP packets containing one complete bundle per packet either from any other BD or only from the remote address specified. [DTN2 does not make any attempt to tie up incoming bundles received over UDP with a specific link or previous hop EID at the convergence layer. The bundle itself may incorporate a *previous hop* block that identifies the previous hop's EID.]

When a link is created associated with an interface using the UDP Convergence layer, the parameters associated with the link (local address, local port, remote address, remote port and rate control parameters) are stored for the link. These parameters will be used later if a contact is opened for sending bundles to the remote address. [In DTN2 the bundles are sent from a different socket than is used to receive. It appears that the local port could be the same as the receiver port which would result in the code being unable to open the sender port. There is also a sending port per contact whereas the receiver port is shared.]

When a contact on a previously created link is opened, a new sending socket is created and is 'connected' to the remote address specified in the link creation. As mentioned above this just fixes the remote address to which packets are sent. There is no handshaking with the prospective remote peer. The sender functionality is called in line in the bundle core thread rather than on a separate thread. The [UDPclayer] document notes that the sending of UDP encapsulated bundles needs to be rate limited to avoid potentially overloading the network on which they are sent. [DTN2 accepts parameters to configure such rate limiting and sets up some capabilities that would allow rate limiting to be provided via a token bucket strategy. However the code does not currently provide for rate limiting the actual output.]

[DTN2 does not implement the proposed UDP keep alive option (Section 2.4 of [UDPclayer].)]

The following functionality is used in the bundle core thread:

325. **Interface up**. Parse the parameters supplied by the creator of the interface. The parameters used by the UDP convergence layer are local and remote IP addresses,

local and remote UDP port numbers, and, optionally, rate control parameters to limit the sending rate on this interface (see item 393 for details of the possible parameters). Create a receiver thread for this interface allocating a receive buffer large enough to hold the largest possible UDP packet. If a specific remote address and/or port are specified the receiving socket will be 'connected' to this address/port pair and the interface will only be able to receive from this location. Otherwise the receiver thread will be able to receive promiscuously from any source address/port pair. Start the receiver thread.

326. **Interface down**. Stop the receiver thread and when it has stopped, delete the receiver thread, closing the associated receiving socket.

327. **Initiate a link associated with an interface on this convergence layer**. Record the convergence layer parameters (see item 396 for details of possible parameters). [DTN2 checks the link's idea of the Maximum Transmission Unit and registers an error if it is *bigger* than the maximum bundle size. Now, OK, the default maximum bundle size is actually the biggest size of UDP payload you could possibly have, but it is not clear to me why you would flag an error if you could carry a bigger bundle.. maybe a warning?] Communication end points and threads are not affected.

328. **Delete link**. If the *link* has an existing contact, close the contact (see item 330)delete any existing contact information from the link. [DTN2 does not close down any existing contact - it will be 'orphaned'.]

329. **Open Contact**. Create a new sending socket based on the parameters registered for the *link*. Extract the remote address and port from the link next hop specification. This must be resolvable locally to a 'real' IP address (not all zeroes) and a non-zero port number. Locally this may include looking up a textual hostname in the local hostname file. If the configuration specifies a sending rate, configure the socket to limit the rate appropriately. See the functionality at item 330). Report an error back to the caller if the socket cannot be created and post a *Link State Change Request* event with the bundle core indicating that the *link* is now unavailable. Otherwise record the sender socket for future use and post a *Contact Up* event on the bundle core.

330. **Initialize UDP sender**. Create a UDP socket for the sender. If an explicit local address and/or port are specified for the contact, bind the socket to the explicit address; otherwise allow the operating system to choose the address and allocate a random port. 'Connect' the socket to the remote address and port specified for the *link*. If rate limiting is requested configure a token bucket mechanism to provide rate limiting [Configured but not used in DTN2.] Allocate a buffer used to hold the wire formatted bundle before sending

331. **Close contact**. Shutdown the sender socket and delete the sender if one is present.

332. **Bundle queued**. Check the *link* and associated *contact* exist and have a sender socket. Use the *send bundle* functionality (see item 333) to transmit the bundle as a single packet. If successful, delete the bundle from the *link send queue*, add it to the *inflight queue* for the link and post a *Bundle Transmitted* event on the bundle core (see items 258, 259 and 57).

333. **Send bundle**. Use the *produce* functionality for the bundle (see item 151) to generate the initial segment of the wire format of the bundle up the maximum allowed size of UDP bundle in the allocated *send buffer*. If the complete bundle is too large to be generated in one buffer length report an error and abandon sending. Otherwise write the send buffer to the send socket as one message. If the whole bundle is not successfully sent, report an error and abandon sending the bundle.

The following functionality is implemented on the Receiver thread:

334. **Receiver thread main loop**. Allocate a buffer large enough to hold the largest possible bundle allowed by the UDP convergence layer. [This is limited by the maximum size of a UDP datagram. DTN2 only implements IPv4 so this is a little less than 2^{16}. Although 'jumbograms' are proposed in IPv6 there has been little movement to actually support them on real networks.] Loop continuously until instructed to stop by the bundle core thread (see item 326). Performing (blocking) reads from the receive socket into the allocated buffer. [I *believe* that DTN2 executes a blocking read and the receiver thread will hang up in this read until a packet arrives or the process gets a signal. This is difficult to really understand because the actual real *recvfrom* system call is buried under several layers of clever Oasys code that might have an associated poll (it ultimately uses the *rwdata* function in io/IOclient.cc). However there is no notifier and no timeout so I believe that this effectively becomes a blocking call. There is a better solution to breaking out of the blocking read involving poll and a second socket that you can send a signal octet down to flip the blocked poll out when you want to stop the receiver. The upshot of this is that deleting an interface will leave the receiver around in a blocked condition. Fortunately deleting an interface is not something you normally do on a production system.] When a packet, which should contain a complete bundle, arrives, pass it to the *process data* function (see item 335).

335. **Process received data**. Create a new bundle and use the *consume* functionality (see item 147) to construct the internal form of the bundle from the received data. The restrictions on the UDP Convergence layer mean that this one chunk of received data should represent a complete bundle. If the *consume* routine does not report that a complete bundle has been constructed, delete the bundle and report an error. Otherwise post a *Bundle Received* event on the bundle core (see item 56). There is no guarantee that the previous hop can be identified. The bundle might contain a Previous Hop block [PrevHopBlock] [DTN2 does not test this]. Otherwise, it would in principle be possible to scan the links associated with this Convergence Layer to determine if the IP address and port that sourced this bundle referred to a link that is configured for sending UDP encapsulated bundles. Failing this the event reports an unknown previous hop and link. [DTN2 does not attempt to do this. This means that using the UDP convergence layer is problematic if you are trying to keep track of where bundles have come from.]

4.9.7.3 Null Convergence Layer

The *Null Convergence Layer* is a convenience and testing facility that acts as a sink for outgoing bundles and has no way of receiving bundles. It is roughly equivalent to the 'bit bucket' device */dev/null* provided on Unix systems. It is a rudimentary implementation of a *Non-connection Oriented Convergence layer* (see Section 4.9.1.2.1).

The *Null Convergence Layer* has one boolean parameter *can transmit* (see item 396). If this parameter is true, bundles are deemed to have been transmitted as soon as they are queued and are place on the *inflight queue* immediately. If it is false, they are left on the *send queue* forever or until the bundle core instructs the convergence layer to cancel the send.

There is no point in creating an interface using the *Null Convergence Layer* as there is no means of sending a bundle to such an interface - it has no connection to an actual network. Thus there is no interface-related functionality but the default functionality will allow an interface to be created and deleted.

The following specific functionality is provided for the *Null Convergence Layer*:

336. **Initiate a link associated with an interface on this convergence layer**. Record the convergence layer parameter (see item 396) for details of possible parameters) and record the value with the *link*.

337. **Reconfigure a link**. The available parameter can be altered if required.

338. **Delete link**. Decouple the parameter value specification from the *link*.

339. **Open contact**. Associate the contact with the link and post a *Contact Up* event on the bundle core.

340. **Handle bundle queued**. If the *link* is configured to specify that it cannot transmit bundles do nothing. Otherwise move the bundle from the *send queue* to the *inflight queue* and post a *Bundle Transmitted* event on the bundle core (see items 258, 259 and 57).

341. **Attempt to cancel a bundle**. If the *link* is configured to specify that it cannot transmit bundles, the bundle should still be on the *send queue* and can be cancelled - post a *Bundle Send Cancelled* event on the bundle core. Otherwise do nothing as transmission will have been simulated already.

4.9.8 Contact Manager

The *Contact Manager* is the central actor in the communications sub-system of the BD. It keeps track of the various links that are in play in the system. The *Contact Manager* is offered most events that are generated in the BD after the bundle core has processed the event, although under some circumstances the bundle core may decide that the event should not or does not need to be processed except by the bundle core. Likewise not all events are of interest to the Contact Manager, and the default ‘null’ processing may be carried out for these events.

The functionality in the Contact Manager can be divided into four groups:

- Link set management. Functionality to manage the set of links currently in existence in the BD. Note that links are essentially never completely deleted. Links that are logically deleted are maintained in an inactive state.
- Event handler routines for a subset of the events in the system (the corresponding functionality in the bundle core is indicated after the event type here.

    - Link Created (bundle core functionality 80)
    - Link Available (bundle core functionality 82)
    - Link Unavailable (bundle core functionality 83)
    - Contact Up (bundle core functionality 76)
- Creation of a new opportunistic link
- Functionality to manage the restart of on demand and always on links that do not have an active contact because they have broken.

A single Contact Manager is created during start up of the BD. When created the Contact Manager creates an empty set of Links and initialises the count of *opportunistic li*nks in existence to zero. The Contact Manager also manages availability timers that try to reopen a failed *on demand* or *always on* link. Each link has a *retry interval* that is initially set to zero. In the case of *on demand* and *always on links*, in the event of a communications failure only, the link is scheduled to reconnect after the current retry interval. The first time this occurs and subsequently when the link Contact Up event occurs (indicating that communication has been (re-)established), the retry interval is set to the configured *minimum retry interval*. If reconnection fails, the reconnection is rescheduled with an interval of twice the current retry interval. Further doublings are made if the reconnection fails repeatedly until the retry interval is capped at the configured *maximum retry interval*. See items 428 and 429 for management commands that are used to set these parameters.

4.9.8.1 Link Set Management Functionality

This section describes the functionality needed to maintain this set of links that are in existence that is stored in the Contact Manager:

342. **Add new link to the *Contact Manager link set***. This routine is primarily used by the management system to record newly created *links* in the *link set*. [The external convergence layer system in DTN2 may also use it but this is not relevant for N4C.] The *link* will already have been created when this functionality is invoked. The *link* is identified by its name. If a *link* already in the list uses the same name, the new *link* is rejected and must be deleted by the caller. Otherwise the new *link* is inserted into the *link set* in the *Contact Manager* and a *Link Created* event is posted on the bundle core (see item 80). [This can be suppressed in some circumstances associated with the external convergence layer by using a *create pending* flag. This is not relevant to N4C.]

343. **Delete a link in the *Contact Manager link set* by name**. The logic for deleting *links* is quite complex. This complexity is needed to avoid deadlocks with nested events and locking the *link set*. Check that the *link* exists and has not already been deleted. When called from the normal management path call the *delete link* functionality for the link itself (see item 249). This removes the convergence layer state information relating to the link and sets the flag indicating that this *link* is in process of being deleted. If the *link* is in state OPEN or state OPENING, post a *Link State Change Request* event on the bundle core for the link to set it to state CLOSED. (see item 84). This results in the *contact* being closed (if it had got as far as being opened) and removed and a *Link Unavailable* being event posted (see items 83 and 348). If there are any *availability timers* associated with the *link*, cancel them. Remove the *link* from the *Contact Manager link set*, and post a *Link Deleted* event for the link on the bundle core (see item 81). The functionality is more convoluted if

called by the external convergence layer rather than the management function. In this case calling the *link delete* functionality must be postponed until the *Link Deleted* event has been processed in the bundle core so that the (external) convergence layer information is still around. In this case the event handler has to be called 'in line' rather than the event being scheduled for later. [N4C does not need this complexity.]

344. **Lookup a *link* based on the textual name of the *link*.** The *Contact Manager link set* is searched for the *link* associated with the name and the *link* returned if it is found.

345. **Search the link set according to various criteria**. The available *links* have their parameters compared against various criteria. The first *link* that matches the specified criteria is returned. The following criteria can be used:

Matching Criterion	Always Match Value
Link type (enumerated)	LINK INVALID
Convergence layer (pointer)	NULL
Nexthop transport address (string)	Empty string
Remote EID	dtn:none
Link states	0xff (all states)

4.9.8.2 Contact Manager Event Handler Routines

346. **Handle Link Created event**. Check that the *link* is not being deleted. If it is being deleted log a warning and do no more. Check that the *link* is in the *Contact Manager link set*. If not log and error and do no more. Otherwise call the *set initial state* functionality for the newly created link (see items 262 and 264).

347. **Handle Link Available event**. Check that the *link* is not being deleted. If it is being deleted log a warning and do no more. Check that the *link* is in the *Contact Manager link set*. If not log and error and do nor more. If an *availability timer* exists for this *link*, delete it - the *link* is now available and the timer is no longer required.

348. **Handle Link Unavailable event**. Check that the *link* is not being deleted. If it is being deleted log a warning and do no more. Check that the *link* is in the *Contact Manager link set*. If not, log an error and do nor more. For *on demand* and *always on* links that have become UNAVAILABLE due to a failed communication link (rather than user action or an idle timeout), create and start an *availability timer*, implementing an exponential back off algorithm for repeated attempts. The timer timeout calls the *reopen link* functionality (see item 351).

349. **Handle Contact Up event**. Check that the *link* is not being deleted. If it is being deleted log a warning and do no more. Check that the *link* is in the *Contact Manager link set*. If not log an error and do nor more. For *on demand* and *always on* links, reset the *retry interval* to the configured *minimum retry interval*.

4.9.8.3 Creation of a New Opportunistic Link

This functionality is used to start a new *opportunistic link* when a connection requests is received at an *Interface* or as a result of a discovery process using a *Connection Oriented Convergence Layer* (e.g., TCP, see item 286).

350. **Create new opportunistic link**. Construct a unique name for the *link* using the opportunistic counter and various link names. Create a new *opportunistic link* using this name and default parameters (see item 406 for a configuration command that provides settings for these defaults) in the *create link* function (see item 248). [It would be useful to be able to set some parameters into the interface to assist here.] Set the remote EID for the link from supplied parameter. Add the new *link* to the *Contact manager link set* (see item 342). Return a reference to the new *link*.

4.9.8.4 Management of On Demand Link Availability

351. **Reopen a link**. Called when the *availability timer* for a *link* expires. Delete the *availability timer*. Check that the *link* still exists in the *Contact Manager link set* and that it is not being deleted. Check that the state of the link is still UNAVAILABLE. If so, post a *Link State Change Request* event on the bundle core seeking to change the link state to OPEN (see item 84). If the state was not UNAVAILABLE log an error [Is this really an error?] probably because there has been a state race with the management function trying to alter the state of the *link*.

4.10 STORAGE MANAGER

The BD is intended to be a long running process that can be stopped and restarted on demand, and will also be able to continue operations in a more or less seamless manner if the supporting platform has to be restarted after a controlled or uncontrolled shutdown (e.g., a power failure). The BD may be configured to automatically shutdown if it becomes idle for an extended period to release processor resources.

Items which need to persist across such a shutdown will be stored under the control of the storage manager. A number of categories of items have to be stored:

- Bundles held in the *all bundles* list in the bundle core.
- Forwarding log information relating to bundles.
- API service tag registrations.
- Link information
- Dynamic Routing State Information (PRoPHET state).

The items stored will be indexed within the category, but the data stored for each item need only be organized so that items can be retrieved as a whole. There is, at present, no need for independent access to each field that makes up the data describing an item. The data associated with each item is therefore serialized into an opaque data ‘blob’ that is stored under the relevant item type and index. In practice individual data items will be created, possibly updated and deleted, but retrieval of data items is confined to BD restart, and in this case all the data items of each type will be read and deserialized in order to recreate the previously stored state.

In addition to the individual items, a number of pieces of global information about the BD state also have to be stored and recreated. These items have to be updated whenever they

change. Included in this information is a data store version number that allows the BD to verify that it can use the stored data. [DTN2 stores a fingerprint of the various sorts of item stored as this version number, which allows the version to be automatically created and verified. This is not essential, but if the data stored is altered the version number must be altered manually otherwise.]

Various sorts of persistent storage can be used. Typically a form of database would be a sensible choice for most of the items, but the payloads of bundles may be very large and are conveniently held separately as operating system files. [DTN2 offers the choice of the Berkeley database, various forms of SQL database or a completely file based system. The details of the storage mechanism are abstracted by the interface described here. The type of storage to be used is set by a configuration command (see item 488). Depending on the type of storage used, a number of additional configuration command are provided to set up the parameters of the storage system (see Section 4.11.7.15).]

The storage manager can be given a storage quota to control the amount of system resources used (see Section 4.11.7.15 and item 495).

The following global functionality is provided:

352. **Initialize a new empty data store**. Set initial values for the global state items.

353. **Open an existing data store**. Verify that the data store version can be accessed by this BD by checking the data store version number.

354. **Re-initialize an existing data store**. Delete all existing data and re-initialize the global state items.

355. **Read the global state items**.

356. **Write and update the global state items individually**.

357. **Flush all the data to persistent storage and close the data store**.

358. **Return the amount of storage used and the amount available for further storage**.

For each of the types of item stored, the following functionality is provided:

359. **Add a new item of the type with a specified key**. Return an error if the key already exists or the data cannot be stored.

360. **Update an existing item of the type with a specified key**. Return an error if the key does not exist or the update fails.

361. **Delete an existing item of the type with a specified key**. Return an error if the key does not exist or the deletion fails.

362. **Return the total number of items of the type in the persistent storage**.

363. **Return a list of the keys of all the items of the type in the persistent storage**.

4.10.1 Bundle Daemon Configuration and Global State

A number of items are stored that record the current state of various persistent store related items, plus values that verify that the format of the persistent store matches the format expected by the BD executable instance that is accessing it. The items stored are:

- A version code for the BD code.
- The running serial number for bundles (*bundle id*). See Section 4.6.2.
- The running serial number for registrations (*registration id*). See Section 4.4.3.
- A version code for the persistent store database format.[DTN2 does this by generating a 'footprint' of the data stored using a magic capability of the Oasys serialization code. This serializes the names and data types of all the items and then generates an MD5 digest of the resulting buffer.]

4.10.2 Bundle State

The key for bundles is the Bundle ID, a 32 bit unsigned integer unique for each bundle created by this BD. The latest Bundle ID is held in the global state and incremented each time a bundle is created. Bundle payloads are held separately in operating system files. The file name is part of the Bundle State.

Associated with each bundle is the bundle's forwarding log. New entries in the forwarding log are created for various events in the system. Forwarding information is primarily associated with links on which bundles were received or forwarded. [DTN2 does not put the forwarding log into persistent store. The reason for this is that there is no persistent naming or identification scheme for links at present. See Section 4.10.4.]

4.10.3 Registration State

The key for bundles is the Registration ID, a 32 bit unsigned integer unique for each registration created by this BD. The latest Registration ID is held in the global state and incremented each time a bundle is created.

4.10.4 Link State

Persistent state should be saved for links in order to be able to correctly store the forwarding log for bundles. DTN2 does not currently do this because it does not have a suitable identification scheme for the links that can persist across restarts. This functionality needs to be developed.

4.10.5 PRoPHET Dynamic Routing State

The key for PRoPHET dynamic routing state is the EID of the remote node for which the information is stored. No global information is required.

4.11 CONFIGURATION AND MANAGEMENT

In the spirit of Internet applications, a BD that is intended to support DTN networking should be managed and configured using communication over the DTN network. [DTN2 uses conventional IP-based communication for remote management.] Thus configuration and management commands should be embedded in bundles that are delivered to an application embedded in the BD which will action the commands and return the results to an external management application in a response bundle.

There are a number of issues that have to be addressed to make this system work effectively::

- Management commands need to be authenticated and integrity protected to protect the BD against subversion by malicious parties.
- Commands may need to be grouped as transactions which are implemented atomically and are actioned as a whole or not at all.
- It may be necessary to restart the whole BD to action certain commands that are only active during start-up.
- Configuration and management commands need to be logged in such a way that an audit trail of configuration changes can be kept.
- A mechanism needs to be provided that will allow a configuration change to be rolled back automatically if it appears that it has caused problems. This could be handled by requiring a positive confirmation to be received within a period of time after the configuration is installed. If the confirmation is not received the configuration change should be rolled back and a message sent to the originator of the change to indicate that the roll back has occurred.

The DTN Research Group is currently (April 2010) working on defining Management Information Bases (MIBs) for various parts of the DTN subsystem and protocols for carrying this information over a bundle based transport. The functionality presented here is still under development as explained in the next section.

4.11.1 Background

Given the remote nature of most expected N4C applications, the goal is to introduce basic functionality for remote management of N4C unit in the field.

The original design idea was to use a NetConf based architecture for DTN2 management. Since DTN2 is relatively new and since there is a strong intent to carry DTN and Bundle technology beyond the fringe area of project like N4C into mainstream application, it seemed most sensible to start with the management techniques currently proposed for the future of the Internet. After extensive investigation, the decision was made to not take this direction at this time. Several reasons contributed to this decision including:

- Lack of manpower and budget within the N4C project to take on such a large development effort, one that involve not only creating the XML schema for DTN2, but would require implementation of secure transport mechanism. Currently the only existing environments for NetConf that are defined are SSH (required but difficult in an RFC5050 environment) SOAP and BEEP. Since none of these would have been appropriate in a DTN, the N4C project was considering the development of a RFC5050 bundle security based transport. Since this requires developing the specification, as well as the code, it eventually became obvious that the project did not have the available talent or budget for such an effort.
- Even if the technical talent and budget could be found among the partners, the time it would take is beyond the time available in N4C, especially if the management is going to be useful during any of the tests.
- The overhead involved in such a system might be more then required in the smaller memory and power budgets available for some DTN applications.

During the same time that the NetConf solution was being researched it became clear that NASA Glenn Research Center was working on a basic SNMP implementation for DTN2. Given the reality of the situation facing N4C, it was decided to use the NASA DTN2 solution for N4C. This code is currently still in development, but is expected to become available in May of 2010. The assumption is that this code can be included in a rebuild of the system shortly after it becomes available.

An SNMP solution is well suited to the N4C project because it has seen a large amount of development and requires much less code space then any NetConf solution. Also, as it is has been in existence for a long time, many off the shelf components are manageable by SNMP MIBS. The primary limitation of SNMP based management is that it is far better suited to monitoring then it is to control. While this is a serious limitation in a DTN project like N4C that serves a remote set of managed nodes, it is a start. Additionally some of the development being done at Ohio University has outlined an approach that may be applicable to N4C as it establishes a subscription model to replace query response model native to SNMP.

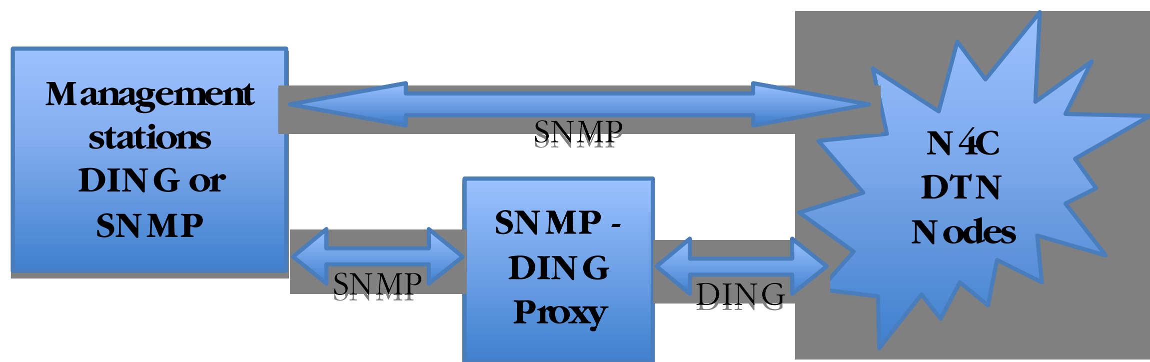

FIGURE 4 MANAGEMENT SYSTEM

4.11.2 Management API

Two different interface options have been developed for the DTN2 SNMP implementation. One uses JSON (Javascript Object Notation) and one uses Diagnostic Interplanetary Network Gateway protocol (DING)[Ding]. N4C has opted to use the DING protocol because of its support of the subscription model.

To quote from the DING specification:

> When a host wishes to receive information about a remote node, the host (called a "subscriber") sends that node (called a "provider") a "subscription request". This request consists of two parts: a "schema" and a "schedule".
>
> The "schema" is a static data model definition that defines exactly which data the subscriber wishes to receive from the provider. This schema consists solely of a list of object identifiers (OIDs). The provider, upon examining the schema, parses this list of OIDs and compares each OID to a list of known OIDs it has. If the OID matches a piece of information it knows how to provide, the provider will transmit that information as instructed by the schedule included with the subscription request. If the provider does not know how to translate a given OID into data, the provider will assign 0 to any field where that OID is requested.

> The "schedule" contains two things: a time interval and a set of conditions. The conditions determine when a provider sends a scheduled frame: the provider will only send a scheduled frame if all conditions are met (the expression provided in the schedule frame evaluates to true). If no conditions are specified, then the provider assumes that there are no conditions to meet and will transmit scheduled frames as determined by the time interval.

The OIDs discussed are those that are defined in SNMP MIBs. While DING has been designed to operate in conjunction with a gateway that runs both the DING client and the SNMP agent, that gateway function can either be included in the same system that runs the SNMP client or in a separate dedicated gateway. Which of these models is used would depend on the environment in which the management is being done, i.e., if the management station is within a DTN zone it would most probably include the gateway function in the Network Management station. The DING gateway functions as a proxy for the SNMP and subscribes to the DING providers in the DTN nodes and responds to the SNMP GET messages. It is not expected that a DING user interface will be provided on a management station, but a simplified CLI could be defined if it was found to be useful for debugging.

In addition to the DING client in the gateway function, it will be necessary for there to be a DING provider in the DTN2 code. As the current DING effort is focused on the ION port of RFC5050 instead of the DTN2 code, it will be necessary for a DTN2 port to be created.

Currently DING is focused on monitoring functionality to satisfy a SNMP GET. Further work needs to be done to support the SNMP TRAP as well as the SNMP SET. Given SNMP's well known weakness as a configuration management tool, further research is still needed on configuration management, but some functionality would be added to the current DING implementation to support the basic settable parameters.

4.11.3 Management Interface

The management interface within the BD is provided by a management internal application (see Section 4.12.3). [This application is not yet available in DTN2 at version 2.7.0 and the functionality is still under development.] In the first instance this application should be used to action commands as documented in Section 4.11, subject to suitable authentication, that were received in a bundle wrapper. The results of the command(s) would be returned in a response bundle. As the DING functionality is developed this application will transition to using the MIB specified in Section 4.11.6. Provision of this application allows the BD to be managed locally using the normal application API or remotely through a remote instantiation of the manager application sending bundles to this BD.

[The initial version of the DTN2 management via bundles can be implemented by passing a TCL script as the payload of a bundle and having it executed by the TCL interpreter embedded in DTN2. the output of the commands can be collected up and returned to the sender of the command bundle as a response.]

4.11.4 Configuration Change Notification

Many of the configuration changes that can be made need to inform the bundle core that the changes have been made. This is done by posting an appropriate event on the bundle core. This is documented for relevant commands in Section 4.11. Note that certain configuration changes will not be applied until the BD is restarted at present.

4.11.5 Logging and Log File Management

The BD is provided with a highly configurable logging system with a dual control system that allows logging to be controlled either at the level of a specific component, instance of a component or through a hierarchical system that organizes the BD functions logically into a hierarchy so that portions of the system covering several related classes can have logging controlled by a single configuration setting, but with the possibility of overriding this setting for more specific portions of the hierarchy. For each class or level in the hierarchy logging can be set to one of the possible levels (in decreasing level of verbosity - *debug*, *info*, *notice*, *warn*, *error*, *critical* and *always*). Logging by class name defaults to *critical* level and hierarchical logging defaults to the level set for the root logger which in turn defaults to *notice* level. In a particular segment of code, log output is produced for messages that are at a level of verbosity equal to or lower than the more verbose of the class and hierarchical setting for the logger in that part of the code.

Debug level logging can be eliminated from production code through a compile time option to reduce the amount of code generated. Efficient macros are provided to reduce the processing cost of logging that is bypassed because of the logging level setting for a component.

All components should contain significant logging to allow operation of the BD to be monitored via the logs.

The logging system is initially controlled via a configuration file but can be adjusted through management commands (see Section 4.11.7.7).

The configuration of the logging specifies a log file, and a scheme for rotating the log file on a scheduled basis to avoid the file growing without limit.

4.11.6 Management Information[2]

The following sections outline the OIDs that have been defined for DTN2 to date. It s expected that further OIDs will be defined by the time the code is ready for test and this specification would have to be updated at that time.

4.11.6.1 Convergence Layer

The convergence layer sits at the bottom of the DTN implementation. It allows multiple networks to be bridged by the Bundle Protocol running above it. The main table is the convergence layer adaptor table. This section deals with interfaces to specific networks, e.g., TCP/IP, mesh, serial, etc.

---

[2] The information contained in the tables in this section are imported from information received from Jim McKim at NASA, DTN-MIB.txt 3088 2009-06-26 in a private communication [DTNmib].

Object	Syntax	Access	Description
claNumber	Unsigned32	read-only	The number of convergence layer adaptors present on this DTN node
claTable	Sequence of claEntry	not accessible	A list of the convergence layer adaptors configured for this DTN.
claEntry	table	not accessible	An entry containing management information applicable to a particular convergence layer adapter.
claIndex	index	not accessible	A unique value, greater than zero, for each adapter. It is recommended that values are assigned contiguously starting from 1. The value has the same semantics as ifIndex in the IF-MIB: the value for each convergence layer adapter must remain constant from one re-initialization of the entity's network management system to the next re-initialization. If an adapter goes away it's index will not be reused, until and unless the NMS re-initializes.
claName	DisplayString	read only	A textual string that describes the convergence layer adaptor. For example, udp-broadcast
claAdminStatus	Integer (1-3) up, down, testing	read-write	he desired state of this CLA. Only CLA's in the up(1) state will send and receive bundles."
claOperStatus	Integer (1-7) up, down, testing, unknown, dormant, notPresent, LowerLayerDown	read-only	The current operational state of this CLA. The testing(3) state indicates that no operational packets can be passed. If claAdminStatus is down(2) then claOperStatus should be down(2). If claAdminStatus is changed to up(1) then claOperStatus should change to up(1) if the adapter is ready to

Object	Syntax	Access	Description
			send and receive bundles; it should remain in the notPresent(6) state if the adapter has missing (typically configuration files) components.
claLinksDiscovered	Unsigned32	read-only	Total number of unique links that have been discovered, and reported, by this adapter.
claActiveLinks	Unsigned32	read-only	Number of links reported by this adapter as currently active. Active links include those that currently support sending bundles and may include links that cannot currently send bundles, for example scheduled links

4.11.6.2 Bundle Protocol

The link table sits right between the convergence layer and the BPA. Each CLA publishes links to the BPA. BPA uses these to forward traffic. Links are considered half-duplex: out only, inbound traffic is not directly measured in the link object. Note: the link table comes from the DTN RI link concept This table is kept logically in the bundle protocol agent section of the MIB even though the metrics are measured and kept by each convergence layer adapter. It was thought that the BPA will always be interested in these statistics and therefore is a good clearinghouse for the data.

Object	Syntax	Access	Description
bpaLinkTable	Sequence	not-accessible	List of the links that are currently known to the BPA. Each link is managed and reported by a single CLA. See the bpaLinkConvLayerIndex. All metrics are determined and maintained by the CLA. This table appears in the BPA section of the MIB because it is assumed that the CLA's will report all link statistics to the BPA. When not otherwise specified all metrics are calculated over the longer of

Object	Syntax	Access	Description
			the lifetime of the link or the time since the management agent was last re-started
bpaLinkEntry	Sequence	not-accessible	An entry containing management information applicable to a link to a remote DTN node. Note: links are considered to be uni-directional from the perspective of the local BPA. Metrics are only kept on the out-going half of the link: A --> B, where A is the local node and B the remote.
bpaLinkName	Display String (1..100)	read only	Textual description of this link. Typically, it includes the name of the destination and is unique among all links with the same source CLA.
bpaLinkState	Integer up, down, unavailable	read only	Current status of the link as set by its CLA. The CLA will set the status to up(1), when the link can actively forward bundles to its endpoint. When bundles cannot be forwarded due to a disruption, the CLA will change the link status to down(2). If the link cannot send bundles due to another reason the CLA should change the status to unavailable(3).
bpaLinkEndpoint	Display String	read only	The URI of the DTN endpoint. This is usually the URI of the remote DTN node itself, e.g., ‘dtn://nodeb’
bpaLinkConvLayerIndex	Interface Index	read only	The convergence layer adaptor that owns this link. The value corresponds to the value of the claIndex field of the claTable.

Object	Syntax	Access	Description
bpaLinkType	Integer ephemeral, scheduled, static, other	read only	Type of link: ephemeral(1) links come and go in an ad hoc manner. scheduled(2) links have an a priori schedule for coming up and going down. See bpaLinkNextUpTime and bpaLinkAvgUpWindow. static(3) links are fixed and are not discoverable. other(4) links covers all types of links not described above.
bpaLinkSpeed	Guage32	read only	The throughput of the link in bits per second. This is the expected throughput of the link when connected and not the current measured throughput.
bpaLinkBundles Delivered	Counter32	read only	The number of bundles that were delivered to their destination. The reporting is based solely upon the convergence layer adaptor responsible for this link; there are no guarantees that these bundles were successfully received by the endpoint.
bpaLinkBundles Expired	Counter32	read only	The number of bundles which have expired while waiting to be sent on this link.
bpaLinkBytesIn	Counter32	read only	The number of bytes of data received as bundles to transmit on this link.
bpaLinkBytesOut	Counter32	read only	The number of bytes of data transmitted out this link.
bpaLinkDelivery Percent	Unsigned32 (0..100)	read only	Percent of bundles sent to this link by the BPA that were successfully transmitted. Measured since the last boot of the management system.

Object	Syntax	Access	Description
bpaLinkContact Attempts	Unsigned32	read-write	For links that use discovery, how many failed attempts are made to contact the remote endpoint before declaring it down.
bpaLinkContacts	Unsigned32	read only	Number of confirmed contacts received by the remote endpoint. In general if this number is equal to bpaLinkContactAttempts, then the link is put in the active state.
bpaLinkBundles Queued	Unsigned32	read only	The number of bundles queued up for transmission over this link. This queue is maintained by the CLA responsible for this link.
bpaLinkBundles Cancelled	Unsigned32	read only	Number of bundles that were canceled by the BPA before delivery.
bpaLinkBytes Queued	Unsigned32	read only	Number of bytes queued.
bpaLinkPercent Available	Unsigned32 (0..100)	read only	Percent of time over the last minute that the link has been active.
bpaLinkUtilizatio n	Unsigned32	read only	Current utilized throughput of this link in bits-per-second. The value is averaged over the last minute. So it would be equal to bytes-out over last minute * 8 bits/byte / 60 bits per second.
bpaLinkNextUp Time	Time Ticks	read only	Predicted time (wall clock time) when this link will next be available. If the link is currently available, this should be the current time. If the next uptime is not known then a value of zero (0) will be returned. Units are 100ths of seconds. The most basic

Object	Syntax	Access	Description
			implementation reports current time when the link is up and zero (0) when the link is down.
bpaLinkAvgUp Window	Time Ticks	read only	The average length of time that this link up contiguously. The average is calculated since the last restart of the management system. This is the expected useful transmit period for this link.
bpaAdminState	Integer up, down, testing	read-write	The desired state of the BPA subsystem. If the BpaOperState is down(2) and the BpaAdminState is up(1), the system should attempt to switch into operational mode.
bpaOperState	Integer up, down, testing, unknown	read only	The current state of operation of the bundle protocol agent subsystem.
bpaExpCheck Period	Unsigned32	read-write	The period, in seconds, between checks for bundle expiration. This determines how frequently the list of outstanding bundles is checked for expiration. Setting it lower will cause the BPA to spend more time looking at each bundle, while setting it higher will cause bundles to remain in memory (queues and bundle store) past their expiration time.
bpaCustodyChec k Period	Unsigned32	read-write	The period, in seconds, between checks for retransmission of custody bundles. This determines how frequently the list of outstanding custody bundles is checked. This is related to bpaRetransmitDelay below.

Object	Syntax	Access	Description
bpaRetransmit Delay	Unsigned32	read-write	The delay, in seconds, before a custody bundle is retransmitted. Retransmission attempts force the router to keep trying to route bundles that are in custody. This value determines how long each bundle goes between retransmission attempts while bpaCustodyCheckPeriod determines how often we check to see if any bundle needs to be retransmitted.
bpaForceStorage	Integer false, true	read-write	If true (non-zero), all bundles will be placed in persistent storage before being routed. If false (zero), storage of bundles is at the router's discretion. This is useful primarily for testing the storage subsystem.
bpaClockSynched	Integer false, true	read-write	If true(1), the system's clock is assumed to be set and properly synchronized across all BPA's in the DTN network. This causes expiration of bundles with absolute time stamps (that is, a non-zero value in the time stamp field instead of a zero-value time stamp and a relative time-to-live) to happen in the usual way. If false (zero), it is assumed that the system does not have knowledge of the current time. Bundles with absolute timestamps will expire only after their lifetime elapses at this node.
bpaUseRel Timestamps	Integer false, true	read-write	If true (non-zero), bundles originating locally will be given a relative time-to-live value instead of an absolute time stamp. The time stamp field of the bundle primary header will be set to zero, indicating that

Object	Syntax	Access	Description
			the lifetime field specifies a time-to-live that will be decreased at every node that forwards the bundle. If false (zero), time stamp and lifetime values work as specified in the Bundle Protocol [RFC5050]. See also bpaTTLInHeader.
bpaTTLInHeader	Integer false, true	read-write	If true (non-zero) and if bpaUseRelTimestamps is also true, the time-to-live value is stored in the lifetime field of the bundle primary header and modified in-place by each node that forwards it. If false (zero) and BPA_FASTPATHUseRelTimestamps is true, an extension block is added to the bundle to carry the remaining time-to-live and forwarding nodes update the TTL in the extension block but not the primary header. This parameter has no effect if bpaUseRelTimestamps is false.
bpaDuplicate Detection	Integer false, true	read-write	If true (non-zero), a record is kept of all bundles delivered to a local endpoint for the lifetime of each bundle in order to prevent duplicate bundle delivery to an application. Duplicate detection imposes a large amount of memory and CPU overhead, especially when many long-lived bundles are in the network.
bpaBundleStorage Capacity	CounterBased Guage64	read only	Number of bytes available for bundle storage; includes free and used space. A value of zero(0) indicates unknown amount.
bpaBundleStorage	CounterBased	read only	Number of bytes of free space

Object	Syntax	Access	Description
FreeSpace	Guage64		available for storing bundles.
bpaLocLatitude	string	read only	
bpaLocLongitude	string	read only	
bpaPendingEvents	Unsigned32	read only	Number of events that the BPA has in its event-queue. Events include: - Bundles received from local applications. - bundles received from local applications
bpaProcessed Events	Counter32	read only	Total number of events processed by the BPA since it was last re-started.
bpaPendingTimers	Unsigned32	read only	The number of timers currently outstanding for the BPA. Timers include - expiration timer for a bundle - re-transmit timer for bundles in the bundle store.
bpaBundlePending	Unsigned32	read only	The number of bundles that await further processing. These include: - bundles received from local applications that have not been forwarded - bundles received from convergence layer adapter that have not been routed to their next hop, or local application. - includes bundles in the bundle store.
bpaBundleCustody	Unsigned32	read only	The number of pending bundles that have their custody bit set.
bpaBundleReceived	Counter32	read only	The number of bundles received by this BPA from the convergence layer. This includes all bundles not sourced by this BPA

Object	Syntax	Access	Description
bpaBundle Delivered	Counter32	read only	The number of bundles delivered to local applications.
bpaBundle Generated	Counter32	read only	The number of bundles received from local applications.
bpaBundle Transmitted	Counter32	read only	The number of bundles transmitted by a link
bpaBundleExpired	Counter32	read only	The number of bundles that have expired prior to transmission or delivery to an application.
bpaBundle Duplicate	Counter32	read only	The number of duplicate bundles received.
bpaBundleDeleted	Counter32	read only	Number of bundles deleted.
bpaBundleInjected	Counter32	read only	The number of bundles injected by the BPA. Typically this only happens in a testing/debugging mode.

4.11.6.3 Routing

Routing in DTN2 is static by default

- Each dynamic DTN routing protocol has an entry in the dtnRouteAlgorithmTable and may provide a reference to its own separately defined MIB.
- For N4C a MIB needs to be created for the PRoPHET router.

Object	Syntax	Access	Description
dtnRouteAlgorithmTable	Sequence	not-accessible	Table of currently configured routing algorithms used by this DTN node.
dtnRouteAlgorithmEntry		not-accessible	Entry for each active routing algorithm.
dtnRouteAlgIndex	Interface Index	not-accessible	Unique value greater than zero (0) for each configured

Object	Syntax	Access	Description
			routing algorithm. This value should remain the same over the lifetime of the management agent.
dtnRouteAlgName	Display String	read only	The name of the routing algorithm.
dtnRouteAlgMib	Object Identifier	read only	A reference to MIB definitions specific to the particular routing algorithm.
dtnRouteStaticTable	Sequence	not-accessible	Table of static routes currently configured on this BPA.
dtnRouteStaticEntry		not-accessible	An entry containing management information applicable to a static route.
dtnRouteStaticEntryIndex	Interface Index	not-accessible	Unique value, greater than zero, for each configured static route.
dtnRouteStaticDest	Display String	read only	Endpoint URI for the destination of the static route.
dtnRouteStatic Source	Display String	read only	Source endpoint URI of the static route.
dtnRouteStaticCOS	Display String	read only	Class of service (COS) that this static route applies to.
dtnRouteStaticNextHopLink	Display String	read only	The name of the link that is the next hop for this static route. The value corresponds to the

Object	Syntax	Access	Description
			bpaLinkName column in the bpaLinkTable.
dtnRouteStaticNextHopCla	Interface Index	read only	The Convergence layer adaptor index of the convergence layer for the next hop. The value corresponds to claIndex in the claTable. A value of zero (0) indicates that the next hop will be determined by another routing algorithm indicated by dtnRouteStaticNextHopLink.
dtnRouteStatic Action	Display String	read only	The action to be taken for bundles matching the src, dest and COS of this static route. Actions include: forward, drop
dtnRouteStatic Priority	Integer low, normal, expedited	read only	The priority to be assigned to bundles forwarded by this static route.
dtnRouteStatic CustodyTimeoutMin	Unsigned32		
dtnRouteStatic CustodyPct	Unsigned32		
dtnRouteStatic CustodyMax	Unsigned32		

4.11.6.4 Info Plane

This will include metrics related to content based routing and the new content based API for applications: publish - subscribe using meta-data descriptions.

Object	Syntax	Access	Description
tbd			

4.11.6.5 Application API

Each of the applications included in the N4C implementation that may require management or reconfiguration, should have it own MIB definitions.

Object	Syntax	Access	Description
tbd			

4.11.7 Configuration and Management Commands

This section specifies the Configuration and Management commands that are available in DTN2 and is rather more DTN2 specific than the more fundamental functionality description. In particular DTN2 is closely bound up with the use of the TCL language and TCL interpreters as a way of controlling the BD. Many of the debugging commands explicitly generate output in a form that can be directly input to a TCL interpreter such as TCL lists, and in some cases actually invoke an embedded TCL interpreter to perform actions.

In DTN2 these commands are invoked through a TCL interpreter. During start-up commands are read from a configuration file and subsequently commands can be given either through a local 'console' interface or, if the BD is running as a daemon, through a remote console interface that passes the commands across a TCP connection.

These mechanisms are not appropriate for use in a future 'production' environment where the BD is likely to be running in an environment where its only communication with the outside world is via a DTN interface. Thus remote management of the BD should be carried out through information carried in bundles using the DING protocol and the MIBs specified in Section 4.11.6; local management should be carried out through generation of management bundles from a local application. The commands detailed here are being mapped and adapted the commands to the set of Management Information Base (MIB) style descriptions defined in Section 4.11.6. The MIBs and the management protocol are being investigated at present in the DTNRG.

Each item in the subsections below describes a command (`set in this typeface`) that can be passed to the DTN2 BD control interface. Optional parameters are generally provided using *`name=value`* syntax for each parameter.

4.11.7.1 API Server Control

These command are used to control the server which implements the primary Application Programming Interface (API) for the BD (see Section 4.4.1). These values only have an effect when the BD is being initialized. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change.

364. **List the settable parameters for the API server controls**.

```
api cmd_info
```

365. **Set/Query flag specifying whether the API server is enabled or not**. Defaults to *true*.

```
api set enabled [true|false]
```

366. **Set/Query the IP address on which the API server will listen for application connections**. Defaults to *localhost* (i.e., 127.0.0.1). Can also be overridden by the environment variable *DTNAPI_ADDR*.

```
api set local_addr [<IP addr>]
```

367. **Set/Query the TCP port on which the API server will listen for application connections**. Defaults to 5010. Can also be overridden by the environment variable *DTNAPI_PORT*.

```
api set local_port [<port>]
```

4.11.7.2 Bundle Control

This section contains a number of commands primarily related to bundles. It also contains some overall BD commands that output BD statistics and set the BD up to close down if it remains idle for a lengthy period.

368. **Inject a bundle into the BD.** This mechanism does not a offer a complete set of options for creating a bundle; it is intended for special purposes and testing:

```
bundle inject <src EID> <dst EID> <payload> <opt1=val1> .. <optN=valN>
```

The *src EID* must be either the null EID or a be the local EID plus a service tag[3]. The *reply-to* EID in the bundle will be the same as the *src EID*. If it is the null EID status reports cannot be returned. The *payload* is a string that is copied into the bundle payload (see *length* option below).

[3] [For bundle injection DTN2 requires that the *src EID* refers to an EID that is used in a registration already in the BD or is the null EID. This constraint was previously applied to bundles to be created using the API function *dtn_send* (see Section 4.4.1.1.12) but it was clear that for general applications it is an inappropriate constraint and it has been removed, although the change has not been reflected in the documentation or comments. However it is still applied here. There is some justification here as injection would normally relate to a bundle injected in response to a bundle previously received, and hence there has to be a registration that can be used as the source EID. Whether this constraint is entirely desirable may be arguable.]

Valid options:
If present set the appropriate bundle processing flag:
*custody_xfer*Request custody transfer
receive_rcpt Send status report when bundle received by a node
*custody_rcpt*Send status report when custody accepted by a node
forward_rcpt Send status report when bundle forwarded by a node
delivery_rcpt Send status report when bundle delivered
deletion_rcpt Send status report when bundle deleted by a node
expiration=<integer>
Set the lifetime of the bundle in seconds. Non-negative integer that
defaults to 60 (seconds).
length=<integer> Force the length of the payload (if longer than the payload string the
payload will be padded out with random octets terminated by a null
octet). Non-negative integer defaulting to zero. If zero, the actual length
of the payload string is used.

369. **Retrieve summary bundle statistics**:

```
bundle stats
```

370. **Retrieve summary bundle daemon statistics**. These include counts of pending events, processed events, and pending timers:

```
bundle daemon_stats
```

371. **Retrieve bundle daemon status**. Intended to check that the daemon event loop is running. Just returns *DTN daemon OK* if the daemon main thread is running. :

```
bundle daemon_status
```

372. **Reset currently maintained statistics**. This clears all the statistics maintained for currently existing links, the summary bundle statistics and the count of events processed in the BD.

```
bundle reset_stats
```

373. **List outline information for all bundles currently known in the BD.** This gives the Bundle ID, source EID, destination EID, length of payload for each bundle and if the bundle is in the *pending queue*.

```
bundle list
```

374. **List the bundle ids of the bundles in the bundle core *all bundles* list**. This should be a complete list of all the bundles in the system:

```
bundle ids
```

375. **Print a detailed dump of the contents of a bundle with a specified Bundle ID (decimal number) as a human readable description**:

```
bundle info <id>
```

376. **Print a detailed dump of the contents of a bundle with a specified Bundle ID (decimal number) as a TCL list**:

```
bundle dump_tcl <id>
```

377. **Print a hexadecimal dump of the payload of a specified Bundle ID (decimal number)**:

```
bundle dump <id>
```

378. **Print an ASCII dump of the payload of a specified Bundle ID (decimal number)**:

```
bundle dump_ascii <id>
```

379. **Force the bundle with the specified Bundle ID (decimal number) to expire now**. This is achieved by posting a *Bundle Expired* event for the bundle for immediate action:

```
bundle expire <id>
```

380. **Attempt to cancel a bundle with a specified Bundle ID being sent on a link with a specific Bundle ID (decimal number)**. This achieved by posting a *Bundle Cancel Request* event for immediate execution. There is no guarantee of success if the bundle has already been sent or is in process of being sent:

```
bundle cancel <id> <link>
```

381. **Empty the forwarding log of a bundle with a specified Bundle ID**. The bundle must on the *pending queue*. This will normally have the effect of getting the bundle delivered or forwarded again:

```
bundle clear_fwdlog <id>
```

382. **Set the BD to shutdown after it has been idle for a period of seconds**. Idle shutdown is disabled by default. [DTN2 has no way to re-disable the idle shutdown without restarting the BD. However it could be set to a very long period.] The parameter is a (signed) 32 bit integer, but negative values would not be helpful:

```
bundle daemon_idle_shutdown <secs>
```

4.11.7.3 DTN2 Console Control

The commands in this section control the BD management console in DTN2. This may be either local or, if the BD is run 'daemonized' i.e., without an attached control terminal, then the console can be run remotely over a TCP/IP RPC interface. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change. Note that if *stdio* is set to *false* and the listening port is set to zero, it will not be possible to use the command interface as no local console will be started and no command server is started. [I believe that the only way out in these circumstances is a signal!]

383. **List settable parameters for the console**:

```
console cmd_info
```

384. **Set/Query flag specifying whether a local console should be spawned talking to stdin/stdout on the BD control terminal**. This has no effect if the BD is being run 'daemonized', i.e., without an attached control terminal. Defaults to *true*.

```
console set stdio [true|false]
```

385. **Set/Query the IP address at which to listen for incoming console connection requests**. Defaults to the local loopback address (127.0.0.1) normally known as *localhost*.

```
console set addr [<addr>]
```

386. **Set/Query the TCP port number on which to listen for incoming console connection requests**. Defaults to zero. This implies that the command server is not started so it is not possible to connect a remote console to the BD. Typically the console port is set to 5050.

```
console set port [<port>]
```

387. **Set/Query the prompt string to display on the console**. Defaults to *dtn%* .

```
console set prompt [<prompt>]
```

4.11.7.4 Discovery Control

Management commands for controlling *Discovery Agents* and associated *Announcements* (see Sections 4.9.1.6 and 4.9.3).

388. **Add a discovery agent to the Discovery Manager's set**. The *discovery_name* is a text string used to identify the discovery agent and the *address _family* selects the type of discovery agent. Currently the BD understands address families *ip*, *bonjour* and *bt*:

```
discovery add <discovery_name> <address_family> [<parameters>]
```

[At present DTN2 only provides for IPv4 transports; IPv6 is not supported.] The

optional parameters that can be supplied depend on the address family.

For the *ip* address family the following parameters are understood:
`<port=integer> [<continue_on_error=true|false> <addr=A.B.C.D> <local_addr=A.B.C.D> <multicast_ttl=integer> <unicast=true|false>]`
where:

port UDP port discovery messages are sent from and received from. This must be specified and needs to be different for each discovery agent if there is more than one using the same local address.

unicast If *true* then the messages are sent as unicast and otherwise they are sent as multicast or broadcast.

addr Remote IPv4 address to which discovery is sent. The usual case is that this is a multicast group address. If not present, the subnetwork broadcast address (255.255.255.255) is used. If *unicast=true addr* must be an IPv4 unicast address.

multicast_ttl The number of IP hops that multicast packets will propagate. Non-negative integer that defaults to one.

local_addr Local IPv4 address used for sending discovery messages. If missing the operating system will choose the interface to use.

continue_on_error Boolean value. If *true*, will allow the Discovery Agent to retry various network operations (such as binding the sending socket or sending an announcement). Otherwise the agent will terminate immediately if there is a network error.

For the *Bonjour* address family there are no optional parameters.

For the *Bluetooth* address family, there is one optional parameter:

local_addr Local Bluetooth address used for sending discovery messages. If missing the operating system will choose the address to use.

389. **Delete a discovery agent identified by a name**:

```
discovery del <discovery_name>
```

390. **Announce the availability of convergence layer services from this BD through a named discovery agent**. The announcement instance is identified by a name, and depending on the convergence layer announced, the address at which the

services are available may also be specified:

```
discovery announce <cl_name> <discovery_name> <cl_type> <interval=N>
[<cl_addr=A.B.C.D> <cl_port=N>] | [<cl_addr=Bluetooth_address>]
```

The common parameters are:
cl_name Name for this announcement (string).
discovery_name Name of the discovery agent to be used to deliver the announcement.
cl_type Name of the convergence layer being announced.
interval Time period between announcements in seconds.
Other parameters depend on the type of convergence layer being announced.

If the *cl_type* is *udp* or *tcp*, then the following optional parameters can be specified:
cl_addr IPv4 address of the interface at which convergence layer services are offered. For *udp* this will be an interface at which bundles will be received. For *tcp* it will be an interface through which connections can be initiated.
cl_port Port number corresponding to the *cl_addr* where convergence layer service is offered.

If the *cl_type* is *Bluetooth* then the following optional parameter can be specified:
cl_addr Bluetooth address of the interface at which convergence layer services are offered.

Announcements are not specified explicitly for the *bonjour* discovery agent.

391. **Remove a named announcement associated with a local discovery agent**:

```
discovery remove <name>
```

392. **List all discovery agents and associated announcements**:

```
discovery list
```

4.11.7.5 Interface Control

Management commands for interfaces (see Sections 4.9.1.3 and 4.9.2).

393. **Add a named interface associated with a convergence layer also specified by name:**

```
interface add <name> <cl_name> [<parameters>?]
```

where:

name Textual name for the new interface.
cl_name Name of the convergence layer to be used

The possible parameters depend on the type of convergence layer used. All are specified by expressions of the form <name>[=<value>]. The value can be omitted for boolean parameters where presence of the name implies a *true* value.

For IP-based interfaces of type *udp*:
local_addr IPv4 address where the interface will accept incoming bundles.
Defaults to allowing the platform to choose the address.
[DTN2 does not support IPv6.]
local_port UDP Port number where the interface will accept incoming bundles.
Defaults to 4556.
remote_addr If specified and other than INADDR_ANY (0.0.0.0) the interface will
only accept incoming bundles from this IPv4 address.
remote_port Interface will only accept bundles sent from this port. Ignored if
remote_addr is not specified or is INADDR_ANY. Must be specified if
remote_addr is specified.
rate Non-negative integer value specifying average sending rate allowed for
interface in octets per second controlled by a token bucket[4].
Defaults to unlimited (0).
bucket_depth Non-negative integer value specifying maximum amount of 'credit' that
can be accumulated in the token bucket in octets. Limits the burst size
that can be sent on this interface. Defaults to unlimited (0). It would
probably be counterproductive to set this limit smaller than the largest
bundle expected to be sent (default 65507).

For IP-based interfaces of type *tcp*:
local_addr IPv4 address where the interface will accept incoming connections.
Defaults to allowing the platform to choose the address.
local_port TCP Port number where the interface will accept incoming connections.
Defaults to 4556.

For Bluetooth interfaces of type *bt*:

---

[4] [DTN2 specifies this rate control mechanism and apparently partially implements it. It does not appear that it is ever activated.]

channel Radio-frequency channel to be used by interface. Allowed range depends
on location. Defaults to channel 10.

For 'Null' interfaces of type *null*:
can_transmit Boolean value specifying whether the interface is able to transmit bundles.

394. **Delete a named interface**:

```
interface del <name>
```

395. **List all the current interfaces and attributes**:

```
interface list
```

4.11.7.6 Link Control

Management commands for adding statically configuring links and controlling all types of links.

396. **Add a statically configured link.** The new link has a specified name, is of a specified type and uses a named convergence layer to reach a specified next hop EID:

```
link add <name> <next hop> <type> <conv layer> [<parameters>]
```

where
name Textual name of the new link.
next hop String specifying the locator (transport address) of the node reachable
over this link that is usable by the specified convergence layer to determine where to send bundles. For example for IP based convergence
layers it would be *host_address:port_number* or *host_name:port_number*.
type String specifying the type of link to be created: the possible types are
ALWAYSON, ONDEMAND, SCHEDULED and OPPORTUNISTIC
conv_layer String specifying the convergence layer to be used: the possible types are
Connection oriented convergence layers:
tcp (TCP over IP), *bt* (RFCOMM over Bluetooth),
serial (connection over a serial link, possibly dial-up);
Non-connection oriented:
udp (UDP over IP),*norm* (NACK Oriented Reliable Multicast over IP),
ltp (Licklider Transmission Protocol over UDP over IP - expected
soon), *null* (DTN's equivalent of the bit bucket or /dev/null),

eth (raw Ethernet frames), *file* (bundles in files),
and possibly others if the external convergence layer is deployed.

Each of the link types and the selected convergence layers may have parameters. These are all specified with a <*parameter_name*>=<*parameter_value*> syntax. The available parameters and values for the various types of link and some of the convergence layers are documented here. At present only the convergence layers used in N4C are covered here (omitted are *serial*, *norm*, *eth*, *ltp* and *file* at present).

Generic link type parameters: There are no specific parameters for any of the different types of links but there are some minor differences in default values (documented in the individual items) and *idle_close_time* must be zero for an ALWAYS ON link:

remote_eid String specifying the remote EID reachable at the remote end of the link.

reliable Boolean requesting reliable delivery if possible when set to *true*.

nexthop String specifying next-hop transport address for convergence layer.

mtu Non-negative integer specifying the maximum size of data transfer unit on the link that will be created (stands for Maximum Transfer Unit). Default is zero which is interpreted as there being no constraint on bundle size. In theory this is used to control proactive fragmentation.[5]

min_retry_interval Non-negative integer specifying minimum number of seconds to wait between attempts to reopen the link. Used when backing off attempts to reopen an ALWAYS ON or ON DEMAND link that has broken. Default 5 (seconds).

max_retry_interval Non-negative integer specifying maximum number of seconds to wait between attempts to reopen the link Used when backing off attempts to reopen an ALWAYS ON or ON DEMAND link that has broken. Default 600 (seconds), i.e., 10 minutes.

idle_close_time Non-negative integer specifying number of seconds a link may be idle before it is automatically closed. Zero means never close and is the default for most link types apart from ON DEMAND where the default is 30 (seconds). This parameter must be zero for ALWAYS ON

[5] [DTN2 does not appear to actually do proactive fragmentation even though there is a function *proactively_fragment* in the Fragment Manager and this link parameter to say when it should happen. See item 167].

links.

potential_downtime
Non-negative integer giving a conservative estimate of the maximum
number of seconds that a link might be expected to be 'down' during
normal operations after a failure. Used by routing algorithms to determine how long to wait before rerouting any bundles queued on a
link that has gone down. Default is 30 (seconds).

prevhop_hdr Boolean value. When *true* indicates that a *previous hop* block should be
added to the bundle before forwarding on this link (see [PrevHopBlock]).
Defaults to *false*.

cost Non-negative integer representing the abstract cost of using this link.
Potentially used by routing algorithms to select optimal path. Default 100.

qlimit_bundles_high and *qlimit_bytes_high*
Non-negative integers specifying high water marks for number of bundles
and number of octets in the *send queue* of the link. Defaults: 10 bundles
and 2^{20} octets (1 Mbyte).

qlimit_bundles_low and *qlimit_bytes_low*
Non-negative integers specifying low water marks for number of bundles
and number of octets in the *send queue* of the link. Used to provide
hysteresis after the *send queue* starts draining before adding more bundles.
Defaults: 5 bundles and 2^{19} octets (512Kbytes).

Parameters for all Connection Oriented Convergence Layers (*tcp*, *bt*, *serial*):

reactive_frag_enabled
Boolean value. If *true* reactive fragmentation should be attempted for
partially received/transmitted bundles. Requires a *reliable* link. Default *true*.

sendbuf_len Non-negative integer used to set the size of the buffer used to assemble
bundle data before it is given to the transport endpoint (e.g., socket) in
octets. Defaults to 32768 (octets). Should be significantly greater than
the stream connection segment size.

recvbuf_len Non-negative integer used to set the size of the buffer used to

receive
bundle data after it is taken from the transport endpoint (e.g., socket) in
octets. Defaults to 32768 (octets). Should be significantly greater than
the stream connection segment size.

data_timeout Non-negative integer used to set the time to wait for incoming data
during each cycle of transmit/receive processing measured in milliseconds
(typically used as the timeout period in a *poll* system call.)
Default 30000 (milliseconds).

test_read_delay, *test_write_delay*, and *test_recv_delay*
[DTN2 test items] Non-negative integers used to configure delays (in
milliseconds) into the system during testing. Defaults are all zero.

test_read_limit, and *test_write_limit*
[DTN2 test capabilities.] Non-negative integers used to limit the amount
of data transferred on each read or write from the transport endpoint.
Both default to zero meaning so not apply any limit.

Parameters for all Stream Convergence Layers (*tcp*, *bt*, and *serial*). These convergence layers follow the transport level protocol defined in [TCPclayer]. Data is transferred in segments on a stream oriented transport layer:

segment_length Non-negative integer specifying the length of a data segment in octets.
The wire format of a bundle to be transmitted is split into segments that
are no longer than this value. This length does not include the segment
header when transmitted. Default is 4096 (octets).

segment_ack_enabled
Boolean value. If *true* send acknowledgements for segments correctly
received. Default is *true*. Forces the *reliable* setting in the Connection
Oriented parameters to *true*.

negative_ack_enabled
Boolean value. If *true* allow the sending of negative acknowledgements
to terminate the sending of a bundle where, for example, resources have
been pre-empted. Defaults to *true*.

keepalive_interval Non-negative integer specifying interval in seconds between keepalive
messages. Defaults to 30 (seconds). Keepalive messages are not

sent
if the value is zero.

Specific parameters for *tcp* convergence layer. Note that the *remote_addr* and *remote_port* are derived from the link *nexthop* parameter for links statically created from this node. For dynamically created links these values will be set from the connection information received either via discovery advertisements or via the parameters from the transport layer when an incoming connection is accepted. There are defaults for these values (INADDR_ANY and 4556):

local_addr IPv4 address to be used for the local end of the TCP connection. [DTN2 does not support IPv6.]

hexdump Boolean value. If *true*, log hexadecimal dumps of incoming and outgoing
data from *sendbuf* and *recvbuf*. Defaults to *false*.

Specific parameters for *bt* convergence layer. For Bluetooth the *remote_addr* only is extracted from the link *nexthop* parameter. It resembles a MAC address - six pairs of hex digits separated by 5 colons. It defaults to the wildcard BDADDR_ANY (all zeroes):

local_addr Bluetooth address to be used for the local end of the RFCOMM connection.

The specific parameters for the *udp* non-connection oriented convergence layer are the same as those used when adding a *udp* interface. However, although the *remote_addr* and *remote_port* can be specified here, they will be ignored and the values from the link *nexthop* parameter used: :

local_addr IPv4 address where the interface will accept incoming bundles. Defaults to allowing the platform to choose the address. [DTN2 does not support IPv6.]

local_port UDP Port number where the interface will accept incoming bundles.
Defaults to 4556.

remote_addr Ignored.

remote_port Ignored.

rate Integer value specifying average sending rate allowed for interface in
octets per second controlled by a token bucket[6].
Defaults to unlimited (0).

bucket_depth Integer value specifying maximum amount of 'credit' that can be accumulated in the token bucket in octets. Limits the burst size that can
be sent on this interface. Defaults to unlimited (0). It would probably be
counterproductive to set this limit smaller than the largest bundle expected to be sent (default 65507).

[6] [DTN2 specifies this rate control mechanism and apparently partially implements it. It does not appear that it is ever activated.]

One specific parameter can be specified for the *null* non-connection oriented convergence layer:

can_transmit Boolean value specifying if the end point can notionally transmit or not.
If *true* bundles are deemed to have been transmitted as soon as they are queued. If *false* bundles are never moved from the *send queue* after being queued for transmission and can only be removed by the use of the *bundle cancel* command (see item 380) or the API *dtn_cancel* routine (see Section 4.4.1.1.13).

397. **Open an existing named link**. Requires that the link is not already open.

```
link open <name>
```

398. **Close an existing named link**. The link must be either already open or in the process of opening.

```
link close <name>
```

399. **Delete an existing named link**.

```
link delete <name>
```

400. **Set an existing named link to be either AVAILABLE (*true*) or UNAVAILABLE (*false*)**. Only an UNAVAILABLE link can be transitioned to AVAILABLE, and only an AVAILABLE link can be transitioned to UNAVAILABLE.

```
link set_available <name> <true | false>
```

401. **Return the state of a named link (UNAVAILABLE, AVAILABLE, OPEN or OPENING)**. This function will never return the transient CLOSING state and the BUSY pseudo-state returns OPEN but the link stats will show that the *send queue* is full.

```
link state <name>
```

402. **Return all available statistics for a named link**.

```
link stats <name>
```

The statistics include:
Number of contact attempts made.
Number successful contacts.
Number of bundles transmitted.

Number of octets transmitted.
Number of bundles and number of octets currently in the *send queue*.
Number of bundles and number of octets currently *inflight*
(being transmitted or awaiting acknowledgements).
Number of bundles cancelled.
Total length of time the link has been ‘up’, including any currently open contact.
An estimate of the throughput during the ‘up’ time.
Any statistics from the router related to the link - for Table Based routers this is the count
of bundles in the *deferred list* queue.

403. **Return a list of currently known link names**:

```
link names
```

404. **Return information about all existing links or one specified link**.

```
link dump [<name>]
```

This includes:
Link name
Next-hop transport address.
Link remote EID.
Link type string.
Convergence layer name.
Current link state.

405. **Reconfigure a named link after initialization**. Note that not all link options necessarily allow reconfiguration but it appears most of the ones in item 396 do allow reconfiguration.

```
link reconfigure <name> <opt=val> <opt2=val2>...
```

406. **Configure convergence layer specific default options for the convergence layer named *cl_name***. The options for convergence layers mentioned in item 396 can have defaults set.

```
link set_cl_defaults <cl_name> <opt=val> <opt2=val2>...
```

4.11.7.7 Log Control

Commands used to control and monitor the BD logging system.

407. **List settable parameters for log control**:

```
log cmd_info
```

408. **Set the pathname of the logfile to be used**:

```
log set logfile <file>
```

409. **Set the pathname of the logging rules file to be used.** This command doesn't result in the rules file being reparsed - see item 414 that forces reparsing:

```
log set debug_file <file>
```

410. **Output a log message string**. Use the logger with path name and at the level specified (one of *debug*, *info*, *notice*, *warn*(*ing*), *err*(*or*), *crit*(*ical*), *always*) to perform this:

```
log <path> <level> <message string>
```

411. **Set the prefix string prepended to each logging message**:

```
log prefix <prefix>
```

412. **Force log rotation now**:

```
log rotate
```

413. **Return the list of logging rules in effect at present**. This generates a list of logpaths and the associated logging level:

```
log dump_rules
```

414. **Reparse the logging rules file**:

```
log reparse
```

4.11.7.8 Miscellaneous Commands

Some miscellaneous useful commands.

415. **Return the help documentation for commands**. With no parameter it returns the names of the command groups that can be used as the first element of a command (e.g., *bundle* or *link*). The list of commands in each of these groups is returned by the full form.

```
help [<group name>]
```

416. **Return the current time of day**. [DTN2 prints this in a very human unfriendly form - seconds and fractions since the UNIX epoch.]

```
gettimeofday
```

4.11.7.9 Bundle Daemon Parameter Control

These commands set a variety of miscellaneous parameters that control how the BD operates. For all these commands, if the final (value) item on the command line is omitted, the current value of the parameter is returned without changing it. Otherwise the parameter is set to the value specified.

417. **List all the settable parameters in this section**:

```
param cmd_info
```

418. **Control deletion of payload files**. Boolean value that controls whether the operating system file that contained a bundle's payload should be deleted when the bundle is deleted (for testing - defaults to *false*):

```
param set payload_test_no_remove [true|false]
```

419. **Control early deletion**. Boolean value that controls whether bundles that have been delivered or forwarded, and which the router does not expect to do anything further with should be deleted immediately rather than waiting till they actually expire, in order to reclaim the resources earlier:. Defaults to *true*.

```
param set early_deletion [true|false]
```

420. **Control duplicate bundle suppression.** Boolean value that controls whether to suppress routing and forwarding of any bundle that is a duplicate of any that are currently in the *pending* list of the BD. Defaults to *true*:

```
param set suppress_duplicates [true|false]
```

421. **Control custody acceptance**. Boolean value that controls whether this BD should offer custody services by accepting custody for bundles that request it via the bundle processing flags. Defaults to *true*:

```
param set accept_custody [true|false]
```

422. **Control reactive** fragmentation. Boolean value that controls whether reactive fragmentation can be carried out if a partial bundle is received or transmitted on a reliable, acknowledged convergence layer. Defaults to *true*:

```
param set reactive_frag_enabled [true|false]
```

423. **Control retry of transmission for unacknowledged bundles**. Boolean value that controls whether completely unacknowledged transmissions on reliable convergence layers should be retried (see item 57). Default is *true*:

```
param set retry_reliable_unacked [true|false]
```

424. **Control randomization of delivery** order. Boolean value controlling a testing facility that allows the delivery order of bundles to be randomized. Defaults to *false*:

```
param set test_permuted_delivery [true|false]
```

425. **Control use of persistent storage for injected bundles**. Boolean value controlling whether injected bundles are held only in memory by default rather than being written to persistent storage. This is a performance optimization. Defaults to *false*:

```
param set injected_bundles_in_memory [true|false]
```

426. **Control default EID type for unknown EID schemes**. Enumerated value used to set the EID type (singleton or multinode or unknown) for the destination EID of an injected bundle if the scheme used by the EID is not known to the BD. Defaults to *multinode*:

```
param set is_singleton_default [unknown|singleton|multinode]
```

427. **Control 'glob' matching for unknown EI schemes**. Boolean value used to control whether 'glob' style matching should be used when comparing against routes and registrations for destination EIDs that belong to schemes that are not known to the BD. Defaults to *true*:

```
param set glob_unknown_schemes [true|false]
```

428. **Control minimum period for connection retry interval**. Non-negative integer value setting the default minimum connection retry interval for links in seconds. Defaults to 5 (seconds). Used in the back off algorithm for determining when next to retry the connection:

```
param set link_min_retry_interval [<interval>]
```

429. **Control maximum period for connection retry interval**. Non-negative integer value setting the default maximum connection retry interval for links in seconds. Defaults to 600 (seconds). Used in the back off algorithm for determining when next to retry the connection:

```
param set link_max_retry_interval [<interval>]
```

430. **Control minimum period for custody timer**. Non-negative integer value setting the default for the minimum period of a custody timer in seconds. Used in calculating the period after which a retransmission should be made if a custody report is not received. Defaults to 1800 (seconds i.e., 30 minutes):

```
param set custody_timer_min [<min>]
```

431. **Control lifetime percentage value for custody timer**. Non-negative integer value setting the default percentage of the lifetime of a bundle that should be used for setting the period of a custody timer. Used in calculating the period after which a retransmission should be made if a custody report is not received. Defaults to 25 (percent):

```
param set custody_timer_lifetime_pct [<pct>]
```

432. **Control maximum period for custody timer**. Non-negative integer value setting the default for the maximum period of a custody timer in seconds. Used in calculating the period after which a retransmission should be made if a custody report is not received. The value zero implies no upper limit. Defaults to 0 (seconds

i.e., no limit):

```
param set custody_timer_max [<max>]
```

4.11.7.10 Registrations Control

As well as facilities for monitoring registrations made by external applications, there are a number of functions that can install specialized registrations primarily for debugging purposes. Registrations are identified by a serial number (referred to as *reg_id* in these commands) allocated when they are added to the Registration Manager.

433. **Delete a registration**: this may have been created by the API. [DTN2 does not protect its internal applications from deletion. Chaos will break out if the registration for the administrative application (see Section 4.12.5) is deleted. Deleting the other internal applications is probably unwise but not catastrophic - deleting the ping echoer (see Section 4.12.1) will mean that the BD does not respond to pings (at least internally although it could be replaced by an external application).]

```
registration del <reg_id>
```

434. **Return a list of all current registrations and some details of their attributes**. These include the registration identifier, whether it is active or passive, the EID associated with the registration the 'failure' action (what to do if a bundle arrives that could be delivered to this registration except that it is currently passive - one of delete (drop) the bundle, store it for deferred delivery and execute a script that will hopefully result in the registration becoming active), in the last case output the script executed (hopefully not *too* long), the expiration period [not actually when the registration will expire which would be more useful] and some information about the subscribe/publish session flags if appropriate.

```
registration list
```

435. **Return a dump of the state of a registration in a format parseable by TCL**.

```
registration dump_tcl <reg_id>
```

436. **For debugging purposes, create a registration that just logs a verbose description of bundles delivered to it together with a representation of the initial part of the payload (up to 128 octets)**. The payload will be logged in ASCII if the characters are all printable and otherwise as a hexadecimal dump. A *Bundle Delivered* event for the bundle is posted on the bundle core.

```
registration add logger <endpoint>
```

437. **For debugging purposes, create a registration that accepts bundles and queues them so that they can be retrieved through the next command (item 438) using TCL**.

```
registration add tcl <endpoint>
```

438. **Execute a TCL command (currently either *get_list_channel* or *get_bundle_data*) in the context of a TCL registration created by the command described in item 437**. For *get_bundle_data,* the oldest bundle received is dequeued and a representation of the bundle as a TCL parseable list is passed to the TCL command interpreter - this can be used as part of a script - and a *Bundle Delivered* event for the bundle is posted on the bundle core. The command *get_list_channel* returns information about the TCL mechanism that passes information about delivered bundles.

```
registration tcl <reg id> <cmd <args ...>
```

where *cmd* can be *get_list_channel* or *get_bundle_data*.
Neither of these commands currently takes any arguments.

4.11.7.11 Routing Control

Generic router control commands. For all *route set* commands, the command returns the current value of the parameter without making any change if the last argument is omitted.

439. **List all settable parameters for the Table Based routers.** Includes Static routers, DTLSR routers and external routers.

```
route cmd_info
```

440. **Query or set the Local EID for this BD**. No default (see item 175):

```
route local_eid [<new_local_eid>]
```

441. **Set Routing Algorithm**. String value specifying the routing algorithm to use in this BD instance. No default. Available types depend on what algorithms have been compiled in to the BD or are available through the external router interface. DTN2 offers *static* (see Section 4.8.3.1), *flood* (see Section 4.8.3.2), *prophet* (see Section 4.8.5), *dtlsr*, *tca_router*, *tca_gateway*, and *external* (last four not yet documented fully here) as router algorithm types.

```
route set type [<type>]
```

442. **Control automatic addition of routes for discovered links**. Boolean value controlling whether or not to automatically add routes for discovered next hop links (see item 209 and Section 4.8.1). Defaults to *true*:

```
route set add_nexthop_routes [true|false]
```

443. **Control automatic opening of discovered** links. Boolean value controlling whether or not to automatically open discovered opportunistic links (see item 194 and Section 4.8.1). Defaults to *true*.

```
route set open_discovered_links [true|false]
```

444. **Set default priority for routes.** Non-negative integer value setting a default priority for new route. Routers use this value to prioritize which route should be used if there is a choice. Defaults to zero:

```
route set default_priority [<priority>]
```

445. **Set search depth for finding *next-hop* via waypoints when routing**. Non-negative value used to constrain the search depth when a router is trying to identify the *next-hop route* to use for forwarding by chaining through a sequence of *waypoint routes* back towards the current node starting from the bundle destination. Defaults to 10 (hops).

```
route set max_route_to_chain [<chain length>]
```

446. **Set subscription timeout**. Non-negative integer setting default timeout for upstream subscription in seconds.
Defaults to 600 (seconds).

```
route set subscription_timeout [<timeout>]
```

447. **Force recomputation of all routes**. [DTN2 does not delete queued bundles from links on which they were queued if the recomputation changes the link that they should be queued on.]

```
route recompute_routes
```

4.11.7.11.1 Static Router Control

The following commands apply specifically when the Static (Table Based) Router is in use.

448. **Add a *next-hop route* to the routing table**. The *dest* is an EID pattern and so may contain wildcard characters. The *link* is a string that is the name of an existing link. See Section 4.8.2 for a description of route types. The actual addition is performed by posting a *Route Add* event on the bundle core (see items 93 and 190).

```
route add <dest> <link> [opts]
```

Various parameters may be specified for each route as *opts*. These are all specified with a *<parameter_name>=<parameter_value>* syntax. The available parameters and values are:
source_eid An EID pattern string specifying that this route should be only be used
for bundles with source EIDs that match this pattern. Defaults to the
WILDCARD_EID (*:*) that matches all sources.
route_priority Non-negative integer used to determine optimum route if there are
alternatives. For default value see item 444.
cos_flags Integer value from 1 to 7 interpreted as a set of three bits indicating that

this route is good for bundles requiring one or more of the three
classes
of service currently defined by Section 4.2 of [RFC5050]. The
route is
good for a class of service if *cos_flags* has the relevant bit set for
the service
i.e., 0x1 for bulk service, 0x2 for normal service or 0x4 for
expedited
service. The default value for this field is 0x7, i.e., the route is
good for all
classes of service.

action Sets the type of forwarding action to either *forward* or *copy*. If
the action is
forward, a bundle destined for a singleton endpoint will only be
transmitted
or queued once on some link and will not use this route if it has
already
been transmitted or queued on some other link. If the action is
copy, a
bundle can be forwarded via this link even if it has been
forwarded on
other links. The default is *forward*.

custody_timer_min Non-negative integer value setting the minimum
period of a custody
timer in seconds. Used in calculating the period after which a
retransmission should be made if a custody report is not received.
For
default see item 430.

custody_timer_lifetime_pct
Non-negative integer value setting the percentage of the lifetime
of a
bundle that should be used for setting the period of a custody
timer. Used
in calculating the period after which a retransmission should be
made if a
custody report is not received. For default see item 431.

custody_timer_max Non-negative integer value setting the maximum
period of a custody
timer in seconds. Used in calculating the period after which a
retransmission should be made if a custody report is not received.
The
value zero implies no upper limit. For default see item 432.

449. **Add a *waypoint route* to the routing table**. The *dest* is an EID pattern and so may contain wildcard characters. The *waypoint EID* is also an EID pattern that identifies a location from which the destination can be reached. See Section 4.8.2 for a description of route types. The actual addition is performed by posting a *Route Add* event on the bundle core (see items 93 and 190). The available *opts* are the same as

those for a *next-hop route* (see item 448) but several of the items are probably not very useful in DTN2. The actual routes that are associated with forwarding are always *next-hop* routes and the custody timer and priority information is taken exclusively from these routes.

```
route add <dest> <waypoint EID> [opts]
```

450. **Delete all routes to a given destination EID pattern**. The deletion is performed by posting a *Route Del* event on the bundle core. [Note the help in DTN2 is wrong… it is not possible with the function provided to delete a single route if there multiple ones.]

```
route del <destination EID>
```

451. **Returns a textual representation of current routing information**. For Table Based routers including static routing it lists all the current route table entries, and any session related information.

```
route dump
```

4.11.7.12 External Router Controls

These management function relate to the optional external router. This is not relevant to N4C. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change.

452. **Set/Query UDP port for IPC with external router(s)**. Defaults to 8001.

```
route set server_port [<port>]
```

453. **Set/Query the interval in seconds between hello messages**. Defaults to 30 (seconds).

```
route set hello_interval [<interval>]
```

454. **Set/Query the external router interface message schema file pathname**. Default is **/etc/router.xsd**.

```
route set schema [<file>]
```

455. **Set/Query flag requiring performance of xml validation on plug-in interface messages**.
Default is *true*.

```
route set xml_server_validation [true|false]
```

456. **Set/Query flag that forces inclusion of meta-information in XML messages**. If true allows plug-in routers to perform validation. Default is *false*.

```
route set xml_client_validation [true|false]
```

4.11.7.12.1 PRoPHET Routing Control

These management controls relate to the PRoPHET dynamic router. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change.

457. **List all settable parameters for the PRoPHET router**.

```
prophet cmd_info
```

458. **Set/Query delivery predictability initialization constant (*P_encounter*)**. Permitted values are double precision floating point numbers between 0 and 1. Defaults to 0.75. See Section 3.3 of [PRoPHET].

```
prophet set encounter [<double>]
```

459. **Set/Query weight factor for transitive delivery predictability calculation (*beta*)**. Permitted values are double precision floating point numbers between 0 and 1. Defaults to 0.25. See Section 3.3 of [PRoPHET].

```
prophet set beta [<double>]
```

460. **Set /Query the factor for delivery predictability aging (*gamma*)**. Permitted values are double precision floating point numbers between 0 and 1. Defaults to 0.99. See Section 3.3 of [PRoPHET].

```
prophet set gamma [<double>]
```

461. **Set/Query the time period scaling factor for the aging equation (*kappa*)**. Non-negative integer giving the number of milliseconds per time unit. Defaults to 100. See Sections 2.1.1 and 3.3 of [PRoPHET].

```
prophet set kappa [<integer>]
```

462. **Set/Query the number of HELLO intervals that must elapse without receiving a HELLO message before the peer is considered unreachable**. Positive integer. Defaults to 20.

```
prophet set hello_dead [<num>]
```

463. **Set/Query the maximum number of times a bundle should be forwarded when using the GTMX forwarding strategy**. Positive integer. Defaults to 5. See Section 3.6 of [PRoPHET].

```
prophet set max_forward [<num>]
```

464. **Set/Query the minimum number of times a bundle should be have been forwarded before being eligible to be dropped when using the LEPR queuing strategy**. Positive integer. Defaults to 3. See Section 3.7 of [PRoPHET].

```
prophet set min_forward [<num>]
```

465. **Set/Query the timer setting in seconds for aging algorithm and Prophet ACK expiry**. Positive integer. Defaults to 180 (seconds). [Does not appear to be used.]

```
prophet set age_period [<val>]
```

466. **Set/Query flag specifying whether this node forwards bundles to other Prophet nodes**. Defaults to *true*.

```
prophet set relay_node [true|false]
```

467. **Set/Query flag specifying whether this node forwards bundles to the Internet domain**. Defaults to *false*.

```
prophet set internet_gw [true|false]
```

468. **Set/Query the lower limit on delivery predictability**. If the delivery predictability drops below this value delete knowledge about a destination. Permitted values are double precision floating point numbers between 0 and 1. Defaults to 0.0039.

```
prophet set epsilon [<double>]
```

469. **Set the queuing policy**. This is used when deciding what bundles to drop if resources become scarce in a node. See Section 3.7 of [PRoPHET].

```
prophet queue_policy=<policy>
```

where *policy* can be one of the following values:

fifo	first in first out
mofo	evict most forwarded first
mopr	evict most favourably forwarded first
lmopr	evict most favorably forwarded first (linear increase)
shli	evict shortest lifetime first
lepr	evict least probable first

470. **Set the forwarding policy**. This is used when deciding what bundles to forward to an encountered node. See Section 3.6 of [PRoPHET].

```
prophet fwd_strategy=<strategy>
```

where *strategy* can be one of the following values:

grtr	forward if remote's delivery probability is greater
gtmx	forward if "grtr" and NF < NF_Max
grtr_plus	forward if "grtr" and P > P_Max
gtmx_plus	forward if "grtr_plus" and NF < NF_Max
grtr_sort	forward if "grtr" and sort description by P_remote - P_local
grtr_max	forward if "grtr" and sort description by P_remote

471. **Set the maximum delay between protocol messages**. The value is a positive integer, in 100ms units, ranging from 1 to 255 (100 ms to 25.5s).

```
prophet hello_interval=<interval>
```

472. **Set the maximum number of routes for Prophet to retain**. This is the number of destinations about which to maintain delivery predictabilities. Set to zero to disable quota. Defaults to 1024.

```
prophet max_route=<number>
```

4.11.7.12.2 DTLSR Routing Control

These management controls relate to the Delay Tolerant Link State Router (DTLSR) [DTLSR]. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change.

473. **Set/Query string value defining the administrative area for the local node**. Defaults to the empty string.

```
route set dtlsr_area [<area>]
```

474. **Set/Query the weight function used in calculating routes in the graph**. Defaults to *estimated_delay*.

```
route set dtlsr_weight_fn [<fn>]
```

where *fn* can be
cost taken from the *cost* parameter associated with a link (see item 396).
delay use actual delay if known. [I am unclear how this is ever set.]
estimated_delay use an estimated delay based on down time record of link.

475. **Set/Query the scaling factor for link weight for links that are down**. Scaling is done by shifting the down time rightwards by the number of bits specified by this factor. Non negative integer in the (useful) range 0 to 32 (larger values will result in a zero weight always). Defaults to zero.

```
route set dtlsr_weight_shift [<shift>]
```

476. **Set/Query the aging percentage for the cost of down links**. Double precision floating point value in range 0 to 100. Defaults to 10.0 (%). [Not clear that this is used.]

```
route set dtlsr_uptime_factor [<pct>]
```

477. **Set/Query flag specifying whether to keep links that are down in the graph but marking them as stale**. Defaults to *true*.

```
route set dtlsr_keep_down_links [true|false]
```

478. **Set/Query interval in seconds before recomputing routes after an LSA arrives**. This is intended to minimize 'route flapping'. Non-negative integer. Defaults to 1 (second). [DTN2 notes that this is not used.]

```
route set dtlsr_recompute_delay [<seconds>]
```

479. **Set/Query the interval in seconds after which routes are locally recomputed**. This has to be done in order to properly age links that are believed to be down. Non-negative integer. Defaults to 5 (seconds).

```
route set dtlsr_aging_delay [<seconds>]
```

480. **Set/Query the interval in seconds between proactively sending a new LSA**. Non-negative integer. Defaults to 3600 (seconds i.e., LSAs are (re)sent once per hour). [DTN2 also has *min_lsa_interval* which is not configurable.]

```
route set dtlsr_lsa_interval [<seconds>]
```

481. **Set/Query the lifetime in seconds for LSA bundles**. Non-negative integer. Defaults to 86400 (seconds, i.e., 24 hours).

```
route set dtlsr_lsa_lifetime [<seconds>]
```

4.11.7.13 Security Control

Commands to manage the security policy database and keys for the Bundle Security Protocol.

482. **Set the security policy for *in*bound or *out*bound bundles**. Specifies the required combination of Payload Security Block (*psb*), Confidentiality Block (*cb*), and Bundle Authentication Block (*bab*):

```
security setpolicy [in|out] [psb] [cb] [bab]
```

483. **Add a key to the Bundle Security Protocol key database**. The *host* string is matched against the host component of a security source or destination URI. The *cs_num* is the identification number of the ciphersuite with which the key should be used. See [BSP] for values. [Should use human friendly strings here.] The key is specified as string with a hexadecimal encoding (two characters per octet) of the key.

```
security setkey <host> <cs_num> <key>
```

484. **Return representations of all the keys in the key database**:

```
security dumpkeys
```

485. **Empty the key database**.

```
security flushkeys
```

4.11.7.14 Shutdown Control

Management control to terminate the BD in a controlled fashion. The persistent data store is closed down cleanly and all communications links closed in a controlled manner.

486. **Shutdown the BD gracefully**.

```
shutdown
```

4.11.7.15 Storage Control

Command to manage the persistent storage used by the BD. Some of these commands are specific to the type of database used for the persistent storage. Most of these values should be set in the BD configuration file which is parsed before the persistent storage access is initialized. Changing these values once the storage system is activated will either have no effect or in a few cases cause the storage system to fail. For *set* commands omitting the last (value) parameter returns the current value of the parameter without making a change. The query version can be used at any time to check how things are setup.

487. **List all settable parameters relating to the persistent storage subsystem**.

```
storage cmd_info
```

488. **Set/Query the storage system to use**. This has to be done early in the initial configuration of the BD. Changing the value of this parameter once initially selected storage subsystem is running will not affect the storage system in use. Defaults to *berkeleydb*.

```
storage set type [<type>]
```

Where *type* is one of:

filesysdb	Store all persistent store items in operating system files.
memorydb	Store all persistent store items in memory only (this is not really persistent!)
berkeleydb	Use a Berkeley Database store (see Berkeley Database web site).
external	Use an external database system via an XML interface.
mysql	Use a MySQL database server.
postgres	Use a Postgres SQL database server.

489. **Set/Query the name of the database to connect to**. Defaults to *DTN*.

```
storage set dbname [<name>]
```

490. **Set/Query the pathname of the directory where the database will be stored**. Defaults to *$INSTALL_LOCALSTATEDIR/dtn/db* (typically */var/dtn/db* or */var/tmp/db*).

```
storage set dbdir [<dir>]
```

491. **Set/Query the flag requesting cold start initialisation of the database.** This is equivalent to the DTN2 dtnd command line argument --init-db. Defaults to *false*.

```
storage set init_db [true|false]
```

492. **Set/Query the flag requesting warm start initialisation of the database**. This is equivalent to the DTN2 dtnd command line argument --tidy. Defaults to *false*.

```
storage set tidy [true|false]
```

493. **Set/Query the length of time in seconds to wait before really doing the tidy operations**. Defaults to 3 (seconds).

```
storage set tidy_wait [<time>]
```

494. **Set/Query the pathname of the directory used to hold payload files**. Default is empty string. When creating storage areas (--init-db or --tidy) will create the last component of the pathname if needed but parent directories must exist. [Could use *mkdir -p* instead.]

```
storage set payloaddir [<dir>]
```

495. **Set/Query storage quota for bundle payloads in octets**. Non-negative 64 bit integer. Defaults to zero which is interpreted as unlimited. Used by routers when deciding whether to accept another bundle.

```
storage set payload_quota [<octets>]
```

496. **Set/Query the number of payload file descriptors to keep open in a cache**. Non-negative integer. Defaults to 32.

```
storage set payload_fd_cache_size [<num>]
```

497. **Report the current storage usage in octets**. [DTN2 currently only reports the storage used by payload files. The database size is ignored.]

```
storage usage
```

4.11.7.15.1 Configuration Settings for File System Database

498. **Set/Query the number of open file descriptors to cache**. Specific to *filesysdb* storage case. Non-negative integer. Defaults to zero (no caching).

```
storage set fs_fd_cache_size [<num>]
```

4.11.7.15.2 Configuration Settings for Berkeley Database

The following settings are only relevant if the storage type is *berkeleydb*. See the Berkeley Database documentation for details of their implications.

499. **Set/Query the flag specifying the use of *mpool* (memory pool) in the Berkeley Database**. Defaults to *true*.

```
storage set db_mpool [true|false]
```

500. **Set/Query the flag specifying the use of database logging in the Berkeley Database.** Defaults to *true*.

```
storage set db_log [true|false]
```

501. **Set/Query the flag specifying the use of auto commit transactions in the Berkeley Database**. Defaults to *true*. This means every operation is separately committed to the database. [Not sure if DTN2 could sensibly do anything else - would probably treat all operations in a session as one transaction?]

```
storage set db_txn [true|false]
```

502. **Set/Query the flag specifying the use of a shared file in the Berkeley Database**. Defaults to *true*.

```
storage set db_sharefile [true|false]
```

503. **Set/Query the maximum number of active transactions in progress in the Berkeley Database**. Non-negative integer. Defaults to 1000.

```
storage set db_max_tx [<num>]
```

504. **Set/Query the maximum number of active locks at any one time in the Berkeley Database**. Non-negative integer. Defaults to 0 - implies we use the Berkeley Database internal default.

```
storage set db_max_locks [<num>]
```

505. **Set/Query the maximum number of active locking threads at any one time in the Berkeley Database**. Non-negative integer. Defaults to 0 - implies we use the Berkeley Database internal default.

```
storage set db_max_lockers [<num>]
```

506. **Set/Query the maximum number of active locked objects at any one time in the Berkeley Database**. Non-negative integer. Defaults to 0 - implies we use the Berkeley Database internal default.

```
storage set db_max_lockedobjs [<num>]
```

507. **Set/Query the period in milliseconds between checks for deadlocks in the Berkeley Database**. Non-negative integer. Defaults to 5000. A zero value disables locking completely.

```
storage set db_lockdetect [<freq>]
```

4.11.7.15.3 Configuration Settings for External Persistent Storage Interface

508. **Set/Query the TCP port for inter-process communication to the external data store**. Defaults to 0 [or maybe 62345?]

```
storage set server_port [<port number>]
```

509. **Set/Query the pathname of the file containing the XML schema for the external data store interface**. No default. [Usually use */etc/DS.xsd*.]

```
storage set schema [<pathname>]
```

4.12 INTERNAL APPLICATIONS

In addition to user applications that can be connected though the Application Programming Interface (see Section 4.4), the BD provides a small number of internal 'applications', primarily for testing and management purposes but also for managing the control plane of routers. The applications here use registrations with reserved identifiers (see Section 4.4.3).

4.12.1 DTN Ping Echo Application

For testing purposes, bundles with a destination to the local EID and a service tag of **ping** will be delivered to this internal application. The application creates a new bundle from the incoming bundle by reversing the source and destination EIDs of the incoming bundle, and sends it out again.

510. Register an active registration for the service tag *<local eid>/ping* with infinite timeout to the internal application.

511. When a bundle is delivered to the registration, create a new reply bundle from the incoming bundle, by reversing the source and destination EIDS. Set the *replyto* and *custodian* EIDs to the *null EID*. Copy the expiration time period from the incoming bundle and use the payload from the incoming bundle. Post a *Bundle Received* event for the replay bundle on the bundle core (see item 56) and a *Bundle Delivered* event for the incoming bundle on the bundle core (see item 58).

4.12.2 DTN Traceroute Application

For testing routing, [RFC5050] provides a mechanism to generate a status report that is sent to either the *replyto* or, if *replyto* is not set, the *source EID* specified in the bundle. This is invoked by setting the **BUNDLE FORWARDED** flag in the bundle processing flags. For further information on what happened, it might be sensible to add a metadata block to the bundle with the next-hop EID used for forwarding the bundle [Not done in DTN2; could use the predefined URI metadata type].

4.12.3 DTN Management Interface Application

Register an application that will accept bundles containing management and control commands that can be used to monitor and control the BD. [This functionality is not provided in DTN2.] In the first instance this application could be used to action commands as documented in Section 4.11 when they were received in a bundle wrapper, subject to suitable authentication; this will be developed to provided configuration and management control through the DING protocol (see Section 4.11.2). The results of the command(s) would be returned in a response bundle. A number of Management Information Bases (MIBs) are under development for the DTN environment and in due course the management interface should be structured to use these MIBs rather then the 'raw' commands. An initial version of the MIBs are described in Section 4.11.6. Provision of this application allows the BD to be managed locally using the normal application API or remotely through a remote instantiation of the manager application sending bundles to this BD.

4.12.4 DTN Heartbeat Application

Register an application that can be configured to send a periodic signal to a specified management application at a remote EID to indicate that the BD is still functioning properly and optionally to provide information about the performance of the BD and the DTN network. It should also be configurable to send the log file periodically so that it can be examined by the manager. [Functionality not provided in DTN2].

4.12.5 Administrative Bundle Handler

Bundles with a destination EID which is just the bare local EID with no service tag are delivered to this internal application. Only administrative bundles should be sent to this address. Of administrative bundles only custody signals should be sent to this handler. The only other sort of administrative bundle currently defined is the status report and that should be sent to a specific user registered service tag.

512. **Register a registration for the local EID with no service tag**. Make it active with infinite timeout and delivery to this application.

513. **When a bundle is delivered to the application, check that the bundle is an administrative record as defined in Section 4.2 of [RFC5050]**. If not log a warning. Otherwise check what sort of administrative record it is. If it is a status record, log an error as status records should not be sent to the administrative address for the node. Otherwise the record should be a custody record. If it is not, log a warning of an unexpected administrative record. [Note: DTN2 has some out of date code that believes there is an announcement type. This is not in [RFC5050]]. Otherwise (the record contains a custody signal), parse the custody signal and if it conforms to the specification in Section 6.1.2 of [RFC5050], post a *Custody Signal* event on the bundle core (see item 97). Otherwise log an error. Finally (in all cases) post a *Bundle Delivered* event on the bundle core (see item 58).

4.12.6 Delay Tolerant Link State Router (DTLSR) Application

The DTLSR exchanges link state advertisements (LSAs) in bundles. These bundles are sent to this application in the BD that is receiving the LSAs from where the payloads are passed to the DTLSR core code for processing. The application registers the service tag *dtlsr* on the local EID.

4.12.7 External Router Application

Control bundles may be sent to and from an external router connected to the external router interface of the BD. This application handles incoming bundles directed to the external router that might be used to pass route control and topology information between BD instances. The functionality provided is highly generic and the structure of the information carried is determined by the XML schema used for the particular external router. [This functionality is not used in N4C is not documented further here.]