



TRINITY COLLEGE DUBLIN
COLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH

I

**Implementing the NetInf Protocol
with HTTP and DTN Convergence Layers
and Using NetInf over DTN
as the Primary Communication Protocol for a Device**

Elwyn Davies



Implementing the NetInf Protocol with HTTP and DTN Convergence Layers and Using NetInf over DTN as the Primary Communication Protocol for a Device

Executive Summary

During the SAIL project the NetInf (Network of Information) ICN architecture was developed and the NetInf Protocol was designed as a way to publish, retrieve and search for information content identified by the ni URI naming scheme that was also developed during the project. Such pieces of information and the ni name that provides an infrastructure-less means for verifying that the name is correctly associated with the information are known as *Named Data Objects* (NDOs). The connection between the ni name and the content is achieved by incorporating a strong cryptographic hash digest and the identifier of the digest algorithm used into the ni name.

It was decided to prototype the NetInf Protocol in parallel with ongoing refinement of the protocol specification. This is in line with the principles of the Internet Engineering Task Force (IETF); this is appropriate because the NetInf Protocol specification has been published as an Internet Draft and the ni URI naming scheme has already been accepted for publication as an RFC.

The NetInf architecture provides for the NetInf requests and responses to be transmitted along paths that span multiple different types of network domain. The architecture proposes to achieve this by using a *convergence layer* architecture where transport protocols are used that are appropriate to the domain across which the NDO is being carried. It was decided that the NetInf Device would most realistically be in a situation where ICN was its sole communication option if it was operating in a Delay- and Disruption-Tolerant Networking (DTN) network. This offered an opportunity to demonstrate the NetInf Protocol operating across domain boundaries. This required that the NetInf Protocol be implemented to operate over suitable transports in Internet domains, where HTTP running over TCP was selected and DTN domains where the DTN Bundle Protocol (BP) was selected.

The work described here covers the implementation of the NetInf Protocol HTTP and BP convergence layers and a gateway node where NetInf messages could be transferred from one CL to the other at the boundary between DTN and Internet domains. The NetInf protocol was then used as the ICN communication medium for the implementation of the NetInf Device, demonstrating how NDOs could be accessed by existing applications and files written on the device can be automatically published using the NetInf Protocol. The DTN CL was implemented using the DTN2 reference implementation to provide the BP infrastructure and transport.

The resulting code base has been made available as Open Source Software and used in other parts of the SAIL project and for testing work in TCD.

This report ends with an analysis of the work that has been done and records a number of observations on the effectiveness and behaviour of the NetInf architecture and protocol together with a number of suggestions for the future direction of work around the NetInf version of ICN.

Table of Contents

1.Introduction.....	4
2.The NetInf Architecture.....	5
2.1High-level View of the Architecture.....	5
2.2Name Resolution & Routing.....	5
2.3Support for Challenged Networks.....	5
2.4Forwarding.....	6
2.5Mobility and Multihoming.....	6
2.6Transport.....	6
2.7API.....	6
3.NetInf Protocol Overview.....	6
3.1 NetInf Protocol Summary.....	6
3.1.1HTTP Convergence Layer.....	7
3.1.2DTN Convergence Layer.....	10
4.NetInf Device Overview.....	18
4.1NetInf Device Operations.....	20
5.Additions and Modifications to DTN2 Code.....	21
5.1ODBC-based SQL Persistent Storage.....	21
5.2Support for Auxiliary Data Table for Bundles in MySQL Database.....	21
5.3BPQ Extension Block Processing and BPQ Cache.....	22
5.3.1BPQ Extension Block and BPQ Cache Changeset 3550.....	23
5.3.2Changes to BPQ Extension Block and BPQ Cache.....	25
5.4Introduction of 'Publication' Status Reports.....	26
5.5Metadata Block Processing.....	27
5.5.1Problems Detected.....	28
5.5.2Improvements and Fixes for Metadata Block Code.....	29
5.6Introduction of JSON Ontology Type for Metadata Blocks.....	30
5.7Completion of SWIG Generated Scripting Interfaces.....	30
5.8Preallocation of BlockInfo Lists and Creation of BP_Local Data.....	34
6.Development of Nilib Python Code.....	35
6.1A Little History.....	35
6.2Licensing	36
6.3Documentation.....	36
6.4Code Volume and Installation.....	36
6.5Design Considerations.....	37
6.5.1NDO Content Size and Digest Generation Efficiency.....	37
6.5.2Multithreaded and/or Multiprocess Clients and Servers.....	38
6.6Overview of Python Nilib Modules.....	39
6.6.1Installation and Installed Scripts.....	39
6.6.2Logging.....	40
6.6.3Core ni URI Support.....	40
6.6.4NetInf Command Line Client Utilities.....	40
6.6.5Server Common Code.....	42
6.6.6Servers Handling the HTTP Convergence Layer.....	44
6.6.7HTTP Server Support Files.....	46
6.6.8Server and Gateway Handling the DTN Convergence Layer.....	48
6.6.9DTN Support and DTN2 BPA Interface.....	50
6.6.10Modified 'Poster' Software.....	50
6.6.11Modified Standard Python Modules.....	51

6.6.12Nlib Setup Support and Documentation.....	52
7.Development of NetInfFS FUSE-based File System.....	52
7.1Introduction to FUSE Filing Systems	53
7.1.1FUSE Installation Notes.....	54
7.2Implementation of NetInfFS using Fuse-Python.....	54
7.2.1Starting and Stopping NetInfFS.....	55
7.2.2Handling Bundle Fragments.....	56
7.2.3Future Optimisation.....	56
8.Analysis and Conclusions.....	56
8.1Demonstrating the NetInf Architecture.....	56
8.2The Rôle of Affiliated Data and Transmitting Alternative Forks.....	58
8.3Multiple Results and Merging Affiliated Data.....	59
8.4Granularity of NDOs.....	59
8.5The NetInf Device.....	60
8.6Practical Utility of the Development.....	60
9.Suggested Further Developments.....	61
9.1NetinfFs Further Work.....	61
9.2NetInf API.....	61
9.3DTN Specification.....	61
9.4NetInf DTN.....	61
Bibliography.....	62
Revision History.....	63

1. Introduction

This document describes work¹ to implement the NetInf (Network of Information) Information Centric Networking (ICN) protocol together with associated application software and demonstrate the software in a device that aims to use the NetInf protocol as its primary external connection. This work has been carried out by Elwyn Davies of Trinity College Dublin (TCD) with input from Stephen Farrell and Aidan Lynch also working at TCD.

TCD has been a partner in the Scalable and Adaptable Internet Solutions (SAIL) EU Framework Programme 7 project during the period 2010-13. The SAIL project was focussed on research into ICN and as part of Work Package B, TCD and other partners have designed the NetInf architecture for the transmission, caching and retrieval of data content items identified by a self-verifying name based on a secure hash digest of the content of the object. The two key specifications that have resulted from this work are

- “Naming Things with Hashes” [FARRELL2012] This specification defines the 'ni' URI (Uniform Resource Identifier) scheme that is used to name the data content items. The combination of the ni name and the content is then called a Named Data Object (NDO). The specification has been published as an Internet Engineering Task Force (IETF) Internet Draft and has been accepted for publication as an RFC (Request For Comments). Publication will be completed once all the documents referenced by the draft have also been published.
- “The NetInf Protocol” [KUTSCHER2012] This specification, which has also been published as an IETF Internet Draft, defines the NetInf protocol which has been developed for publishing, retrieving and searching for NDOs, especially those identified by ni URIs. The protocol uses a 'convergence layer' architecture to allow in to function inter-domain across multiple different domain types.

Part of the work that TCD committed to carry out in the SAIL project was the development of a 'NetInf Device'. The intention here was to determine the extent to which a user device such as a netbook or tablet computer could operate with ICN as its sole external communication mechanism. To make this situation somewhat more realistic and also to motivate the development of a convergence layer for NetInf that will operate in a DTN (Delay- and Disruption-Tolerant Networking) scenario, the NetInf Device is limited to communication through DTN.

The work described here provides the infrastructure that gives a NetInf capability on the NetInf Device and allows NDOs to be transmitted across both Internet-like well-connected domains using HTTP as the convergence layer protocol and across communication challenged domains using the Bundle Protocol [RFC5050]. The TCD implementation of the NetInf protocol was carried out primarily in the Python scripting language, but early work was also done both in C and, as a PHP plug-in for the Apache web server. Other SAIL partners have developed interoperable implementations of the HTTP CL using other languages and also a UDP CL.

Sections 2. and 3. give an overview of the architecture of the NetInf approach to ICN and details of the NetInf protocol developed to support it, including a first description of the NetInf CL using the DTN BP. Section 4 describes the NetInf device architecture. Section 5. covers the modifications made to DTN2 to support the NetInf CL and Section 6. describes the components that make up the Python part of the Nilib which provides an implementation of the NetInf protocol with HTTP and BP CLs. Section 7. describes the NetInfFS developed for the NetInf Device. Finally Section 8. analyses the outcome of the work and describes some areas of further research and development that seem to be indicated as a result of the work described here. Section 9. gives an outline of some immediate refinements that could be carried out and records some issues either DTN2 that have come to light during the work but have not been resolved.

¹ This work was partially supported by the Scalable and Adaptable Internet Solutions (SAIL) EU Framework Programme 7 project (Grant # 257448) and by Folly Consulting Ltd.

2. The NetInf Architecture

2.1 High-level View of the Architecture

The following statements describe the NetInf architecture at an abstract level.

- NetInf enables access of named objects, and defines a naming scheme for these objects. The NetInf naming scheme is designed to make objects accessible not only via NetInf protocols, but also via other ICN protocols.
- The NetInf layer performs routing and forwarding between NetInf nodes based on NDO names.
- NDO names are independent of the location of the object in the network topology.
- Caching of NDOs may be provided at any node in a NetInf enable network.
- A multi-domain NetInf network may include domains with challenged networks, such as Delay- and Disruption-Tolerant Networking (DTN) networks.

An overall view of the construction of a NetInf enabled network is shown in Figure 1.

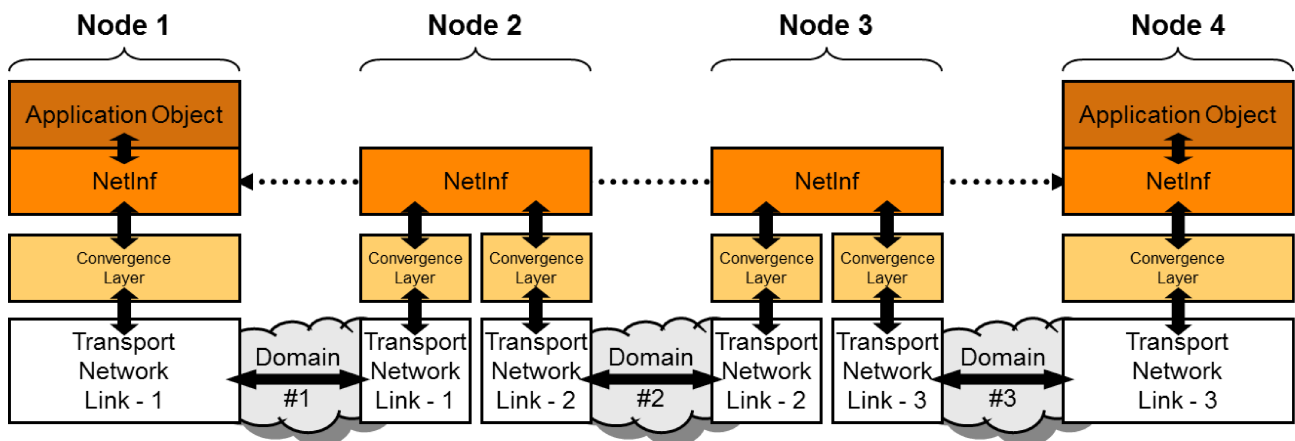


Figure 1: NetInf Architecture

2.2 Name Resolution & Routing

NetInf may perform routing based on the names of NDOs. However, routing based on flat names from NDOs may not scale in a global NetInf network. Therefore, the global NetInf network may use an NRS to map these names to locators that will identify physical entities of an underlying network, in order to take advantage of global routing and forwarding in this underlying network. An example of such an underlying network is the current Internet. This allows for scalability, because the routing and forwarding plane has only to cope with the network topology, and not with the location of single NDOs. The underlying network can be any interconnection of heterogeneous L2 or L3 subnetworks. There is (at least) one NRS for the global NetInf network (interdomain name resolution). There may be name resolution systems which are local to a domain or a host (intradomain name resolution).

2.3 Support for Challenged Networks

In some cases (e.g., in a challenged network domain) it is not possible to resolve NDO names to locators of an underlying network at the source. However, the resolution may be performed by an intermediate NetInf node along the path to the destination (late binding). In this way, challenged network domains may act as NetInf domains with their own routing and forwarding strategies.

2.4 Forwarding

In order both to improve scalability and to cope with situations where a NetInf layer message has to be forwarded across network domain boundaries, NetInf messages will often be forwarded incrementally through several NetInf hops, where a NetInf hop might be made up of several hops in the underlying network. By using identifiers or locators that refer to network structures at various scales, aggregation can be used to control the amount of state needed for NetInf forwarding. Messages are initially directed to gateways in large scale structures which can incrementally refine the direction of the message.

2.5 Mobility and Multihoming

In a global NetInf network, mobility and multihoming are based on dynamic updates of the bindings in the Name Resolution System (NRS) between the NDO names and the identifiers or locators used for forwarding. Alternatively, mobility and multi-homing may be based on dynamic updates of the NetInf layer routing information, so that the NetInf routing system announces the current location of NDOs.

2.6 Transport

NetInf interconnects a variety of networks with different address spaces, different network technologies, and different owners. There is a Convergence Layer (CL) which adapts the NetInf layer to different types of underlying networks on a per hop basis. Examples of such underlying networks are TCP/IP networks, Ethernet, DTN networks using the Bundle Protocol, etc.

2.7 API

The NetInf Application Programming Interface (API) is NDO-oriented as opposed to a classical channel-oriented and host-oriented API such as the socket API for TCP/IP. This means that in the NetInf API, NDOs are addressed directly by their names.

3. NetInf Protocol Overview

NetInf employs one conceptual protocol providing accessing named data objects as a first-order principle. Thus there is one simple protocol that all nodes implement. The NetInf protocol is message-based; it provides requests and responses for PUBLISHing, GETting, and SEARCHing for NDOs. These requests and responses employ the common naming format and the common object model.

3.1 NetInf Protocol Summary

The specification of the NetInf protocol that is intended to be used to realize the NetInf architecture described in Section 2. has been published as an Internet Draft [KUTSCHER2012]. This section contains a brief summary of the protocol as described in the Internet Draft and has been implemented as part of the work described in this document.

The CL concept on which the protocol is based has been realized in a number of ways using the Hypertext Transfer Protocol (HTTP) over the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP) and the DTN Bundle Protocol (BP) as transports for NetInf messages. The abstract protocol and the CLs that have been developed are all based on the use of three pairs of messages and responses:

GET/GET-RESP

The GET message is used to request an NDO from the NetInf network. A node responds to the GET message if it has an instance of the requested NDO; it sends a GET-RESP that uses the GET message's msg-id as its own identifier to link those two messages with each other.

PUBLISH/PUBLISH-RESP

The PUBLISH message allows a node to push the name and, optionally, a copy of the object

octets and/or object meta-data. Whether and when to push the object octets vs. meta-data is a topic for future work. Ignoring extensions, only a status code is expected in return.

SEARCH/SEARCH-RESP

The SEARCH message allows the requester to send a message containing search keywords. The response is either a status code or a multipart Multipurpose Internet Mail Extension (MIME) object containing a set of meta-data body parts, each of which MUST include a name for an NDO that is considered to match the query keywords.

The combination of the NetInf protocol layer with choices of appropriate CLs offer a number of advantages over existing Content Delivery Network (CDN) approaches:

- transport of the content can be controlled from the receiver;
- applications can receive additional information (metadata) about the content;
- content can be accessed across domain boundaries where different network technologies are in use (e.g., DTN and IP); and
- transport can be optimized for the network type and conditions that prevail (e.g., coping with intermittent connectivity and mobility).

Routing and selection of the next-hop destination for request messages is effectively shared between the NetInf layer and the convergence layer. The NetInf layer can, depending on local policy and knowledge of the local network environment, choose to select a specific next-hop node, a set of next-hop nodes or use a multi-cast or broadcast technique to have the request sent to, for example, all the nodes on the local subnetwork. Routing and forwarding of the request to the specified next-hop(s) is subcontracted to the convergence layer which uses whatever means is appropriate to the CL to direct the request to the next-hop(s) specified where the NetInf layer attempts to action the request locally and/or select further routes and next-hop(s) for the request.

Responses to requests are reverse path routed. This means that NetInf nodes have to keep state to allow them to match responses with requests previously forwarded (using the *msg-id* in the corresponding messages) and direct the responses towards the source of the request. A request may result in multiple response that may be merged if they are received at a common node on the path.

The published specification [KUTSCHER2012] contains outline specifications for both the HTTP and the UDP based CLs. The object model for the CLs makes extensive use of MIME and JavaScript Object Notation (JSON) has been used to encapsulate any affiliated data that is carried in CL messages.

I have also developed a specification for the DTN BP-based CL which also makes use of the JSON scheme to encapsulate affiliated data carried by the NetInf messages.

The following Sections 3.1.1 and 3.1.2 give an outline of the HTTP and DTN CLs.

3.1.1 HTTP Convergence Layer

The HTTP CL implements the NetInf protocol layered over the HTTP protocol running over a TCP connection. HTTP is designed for use in a client-server paradigm so the CL implementation provides a unidirectional transport for NetInf messages from a client node that originates messages (GET, PUBLISH or SEARCH) that are sent to a server node that manages a cache of NDOs. The server processes the received message and sends appropriate responses back to the client over the same connection. If the node running the client also wishes to maintain a cache of NDOs that can be accessed via the NetInf protocol, it will also have to run an HTTP server that can accept NetInf messages from other client nodes. The situation for a symmetric arrangement of two nodes is shown in Figure 2. If appropriately implemented the server component can handle connections from an arbitrary number of other nodes in parallel, subject to system constraints.

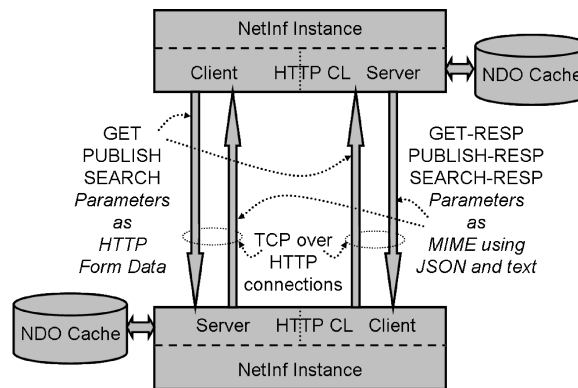


Figure 2: NetInf Protocol HTTP Convergence Layer

The server can also contain a routing component which can forward requests to other nodes if policy is appropriately setup and the node has access to appropriate NRS capabilities. The forwarding can use either HTTP or any other CL for which the node has capabilities depending on the next hop to which the request is to be forwarded. Reverse routing just requires responses to be sent in the reverse direction along the TCP connection where the request was received.

The combination of HTTP over TCP used for the connections provides all the capabilities for a NetInf CL specified in Section 3 of the NetInf protocol specification [KUTSCHER2012] without the CL having to implement any additional facilities, since TCP provides reliable in-order delivery of messages and HTTP provides the necessary message boundaries. The rest of this section provides an outline of the HTTP CL; for more details please refer to the full specification given in Section 6.1 of the protocol specification [10].

NetInf requests are encoded in HTTP forms and Multipart MIME HTTP response bodies are used for the corresponding NetInf responses where there is more than one piece of information to return. The requests are sent using an HTTP POST operation so that the parameters, including the content in the case of the PUBLISH operation, can be sent with the request. HTTP offers two choices for encoding form data to go with POST operations. Either *application/x-www-form-urlencoded* or *multipart/form-data* can be used and the server should be capable of accepting either type for all operations. However in the case of a PUBLISH request carrying the content octets, the use of *multipart/form-encoded* is mandatory because of the potential large size of the content.

These choices have been made because they allow web browsers to interact easily with NetInf and because there are many tools available that make implementation relatively easy. However, the HTTP CL is also intended for use between NetInf 'infrastructure' nodes without human users as well as for conventional web servers and browser clients. Implementations have therefore been constructed that use web browsers and forms as the front-end, integrated with a web server (Apache), and also as stand-alone components for use in non-interactive applications.

NetInf also expects that HTTP CL servers will support accessing the NetInf cache directly using an HTTP URL (rather than an ni URI) using a mapping of the ni name to a URL where the path component starts with '.well-known/ni'. This is equivalent to a NetInf GET request. Due to the advisory constraints on '.well-known' URLs, the server should not send back the content directly if it has it, but should send a redirect to a locally understood HTTP URL that will retrieve the content but does not include '.well-known' in the path of the URL.

Extensive use of JSON objects is made to provide relatively compact encoding of 'affiliated data' that is carried by the NetInf protocol messages. The term 'affiliated data' is used in the protocol specification to cover both 'metadata' that describes attributes of the NDOs being transmitted and metadata of the protocol. Any JSON objects carried consist of sets of unordered name/value pairs at the top level. As is normal with JSON, the values themselves may be arbitrarily complex nestings of objects or arrays of objects.

If the server is unable to successfully process a message for any reason, it returns an HTTP error response with a suitable error code. An HTTP 200 response is sent when processing is successful together with a response body that is described in the following sections., with more details contained in the full specification [KUTSCHER2012].

3.1.1.1 GET operation

The form for the **GET** message has two mandatory fields:

URI the name for the NDO to be retrieved, typically an ni: scheme URI [FARRELL2012].

msgid the message IDentifier (ID) (must be unique per CL hop and request/response pair).

and one optional field:

ext Provided for future extensions and not currently used in our implementations.

The form parameters are relatively compact so

The format for a **GET-RESP** message depends on whether the server is returning both the content octets of the NDO and its affiliated data, or just the affiliated data if the server's cache does not contain the content octets and the request has not been propagated further. In the second case the response body consists of a single MIME component of type application/json encoding the affiliated data. If both parts are returned, the response body is a two part multipart/mixed MIME type, consisting of the affiliated data as before and the NDO content using the appropriate MIME type for the content.

The affiliated data JSON object can contain the fields described below. Note that this remains a work-in-progress and is liable to change as the implementations mature and are further developed so the reader should expect a few minor inconsistencies between this description and the Internet-Draft and the code. As one might expect, the code is at the bleeding edge.

NetInf String describing the version of NetInf protocol in use (e.g., "V0.2").

ni The ni: scheme URI for the NDO retrieved.

msgid Copy of the msgid from the GET message that resulted in the response.

ts A timestamp. There are no fixed semantics for this as it is mainly for debugging, currently this contains the time at which the response was generated in most cases.

loc A list of locator names from where the NDO might potentially be retrieved.

metadata A JSON object containing content metadata recorded with the NDO when it was published plus information about how it was published.

Some implementations (including this one) also return additional JSON elements as listed below.

status A code, taken from the HTTP 2xx success response codes indicating what has been returned.

ct The MIME content type of the NDO content, if known.

searches An array of JSON objects describing searches that have flagged up this NDO as matching the search criteria.

3.1.1.2 PUBLISH operation

The form for the **PUBLISH** message has the same mandatory fields (URI and msgid) as the GET message (see Section 3.1.1.1) with some additional optional fields:

loc1/loc2 Locators where the NDO might potentially be retrieved.

fullPut Boolean value indicating if the publication operation includes the content of the file or just metadata.

octets	If fullPut is true, this form field provides the content of the NDO and information about the type of the content.
rform	Allows the user to choose a browser friendly Hypertext Markup Language (HTML)-encoded report on the publication or a JSON encoded report suitable for automated processing and/or sending through additional CL hops.
ext	A JSON encoded object that can contain a meta field. The meta field is expected to contain a JSON object that is stored as metadata for the NDO content on the server whether or not the content itself is stored. In future, additional fields can be added to the ext object.

If the server receiving the **PUBLISH** message does not have this NDO stored already, and both policy and resources allow, the server will add the metadata and the content (if supplied) to its cache. Subsequent **PUBLISH** messages for the same NDO can be used to add the content if the first publication did not supply it or just to add additional or updated information to the affiliated data.

The affiliated data stored for an NDO records information about the NDO needed to fill in the JSON response object described in Section 3.1.1.1. This object is sent back as the response body if the user selects the JSON response format. Otherwise a web browser (and human) friendly report is sent back encoded as HTML.

3.1.1.3 SEARCH operation

The form for a **SEARCH** has two mandatory fields, a **msgid** field as used in the GET message plus the tokens field described here:

tokens A search query string appropriate for the search mechanism used, and optional fields **rform** and **ext** as used in the PUBLISH form (see Section 3.1.1.2).

On receiving a **SEARCH** message, the server carries out the search using the selected mechanism and returns a response body, in the format selected by the user using the **rform** parameter, enumerating the ni: scheme names for the NDOs flagged by the search mechanism and additional information as appropriate. The search may be carried just among locally cached NDOs or extended to other nodes according to local policy at the server.

3.1.2 DTN Convergence Layer

The DTN Convergence Layer implements the NetInf protocol layered over the DTN Bundle Protocol (BP) [RFC5050]. The BP may in turn be layered over any of its convergence layers depending on the type of underlying network (e.g., the TCP convergence layer or the LTP convergence layer). To avoid any possible confusion the acronym DTNCL will be used if there is any occasion to refer to the underlying DTN convergence layers in this document, whereas the unqualified CL will refer to a NetInf convergence layer.

Applications using the DTN CL obviously need to be aware that responses to NetInf messages are unlikely to be delivered with the same sort of alacrity that might be expected from the HTTP CL running on the well-connected Internet. Programs and users should not 'block' waiting for near-immediate responses but should be prepared for such responses to return maybe hours or even days after the request is injected into the DTN network. Accordingly applications may need to provide persistent storage for the message identifiers used for request messages in order to match them with responses received effectively asynchronously at a much later time.

The NetInf CL layered over the BP uses a specialized extension block known as the Bundle Protocol Query (BPQ)extension block. This block has been developed to support the transport of the additional identifying information for NDOs. An initial version of the specification of the block has been published as an Internet Draft [BPQ2012]². In addition to the standard primary, payload and BPQ blocks, the NetInf CL also uses a

² A number of necessary improvements to the specification have been identified and it is expected that an updated version will be published shortly.

Metadata extension block to carry any additional affiliated data and may use an additional placeholder Metadata block to indicate that the content of the NDO is not included in the bundle.

The BPQ extension block is intended for more general application than just in conjunction with NetInf; It provides a more general mechanism that would allow an application to query the contents of the bundle cache associated with the Bundle Protocol Agent (BPA) at any node that a (query) bundle passes through whilst being forwarded in a DTN network. Currently it is intended that only cached bundles containing BPQ blocks would be examined to determine if they satisfied the query. The BPQ block contains the following fields to support this operation:

kind A BPQ block can be associated with a query (*kind* = 0), or a response to a query that may (*kind* = 1) or may not (*kind* = 2) be fragmented during transmission. This has been extended beyond the specification in version 00 of the draft to allow a BPQ block to be associated with the publication of an NDO (*kind* = 3).

matching rule type

The operation that is performed to determine if a bundle in the BPA cache matches the query represented by a bundle with a BPQ block with a query *kind* field. The specification allows for various different matching rules to be defined. Type 0 specifies an exact match between the *query* field in the query bundle and the *query* field; this is the only matching rule defined in the specification. An additional matching rule (type 1) is used in the NetInf CL to provide a partial match between search tokens for NetInf **SEARCH** requests and a further (non-)matching rule (type 2) is used in **PUBLISH** response bundles which requests that no matching is done. This avoids trying to cache **PUBLISH** responses which are not useful. According to the BPQ specification, the *matching rule* field in bundles matched against cached bundles must have the same *matching rule* in the *query* and the cached bundle. The extra *matching rule* values allow for separate matches against cached NDO bundles for **GET** messages and cached search result bundles for **SEARCH** messages.

creation timestamp

In responses this field records the creation timestamp and sequence number of the bundle that matched the query, as opposed to the creation information for the response bundle that refers to the time when the BPQ query match occurred.

original source endpoint ID (EID)

In responses this field records the source EID of the bundle that matched the query, as opposed to the source EID for the node that generated the response bundle

query string

In queries the string to be matched with the corresponding field in the BPQ block of cached bundles according to the selected matching rule in the query.

query ID An application selected random string placed in queries and copied into the BPQ block of any generated responses (missing from version 00 of the specification).

fragment list

A list of fragment offset/length pairs that can be included in a query to specifically request part of the bundle payload. A node will only return the whole bundle or fragments that contain some or all of the segments requested if it has matching fragments. If the node is unable to satisfy the whole request, it may modify the fragment list in the query to remove any items that it has 'satisfied' by generating a response that contains that fragment.

All bundles involved in the NetInf DTN CL carry a BPQ block. The details of how the BPQ blocks are used are covered in Sections 3.1.2.1 to 3.1.2.4.

As with the HTTP convergence, the affiliated data is mostly carried in a JSON object encoded as a string. For all NetInf DTN CL bundles that carry affiliated data, the JSON string is carried in a Metadata extension block. Initially a Metadata ontology type code from the experimental range has been used to identify the

JSON carrying Metadata block. It is expected that a formal specification of the ontology will be created and submitted to IANA to request the allocation of a permanent ontology type in due course.

All bundles matching the BP specification have a payload block even if it has zero length. Since in principle an NDO can have zero length content, it would be necessary to make components aware of the digest strings in the various possible ni URLs for zero length content in order to distinguish absence of content from actual zero length content. To avoid this, NetInf GET response bundles which are carrying only the affiliated data and not the NDO content will carry a 'placeholder' Metadata extension block which signifies the absence of the content octets. The ontology type for this will be taken from the experimental range initially. The content of the block will be an arbitrary string that can be used however the node feels appropriate. Consideration will be given to submitting a formal specification of this block type but there may be alternative ways to handle the passing of the required state information (e.g., by embedding it in the JSON metadata – a different formal specification would be needed for this alternative).

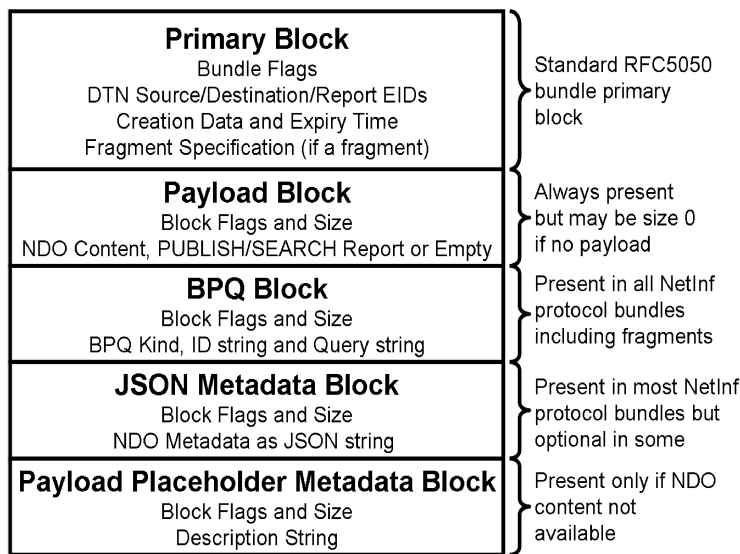


Figure 3: General Block Structure of NetInf DTN CL Bundles

The following subsections describe how the NetInf messages are mapped onto the BP bundles using the BPQ block and the two types of Metadata block just described. The general structure of the bundles is shown in Figure 3. As with all BP bundles, the bundle header may specify that status reports of various kinds should be returned by any BPAs that process the bundle. Although it is not forbidden to request custody transfer for a NetInf message bundle, using custody transfer is probably inappropriate for NetInf messages; this is discussed further in Section 3.1.2.4 where the routing of NetInf messages is discussed.

3.1.2.1 GET operation

The **GET** message is carried in a bundle with BPQ block, a zero length payload and an optional JSON Metadata block. The BPQ block is a *query* block (kind = 0) and carries the two mandatory parameters for the **GET** message:

- URI** the name for the NDO to be retrieved, typically an ni: scheme URI [FARRELL2012] is carried in the *query string* field..
- msg-id** the message IDentifier (ID) (must be unique per CL hop and request/response pair) is carried in the *query ID* field.

The *matching rule* in the *query* block is 0 (exact match required) so that responses will contain the specified **URI** exactly.

Because the **GET** message has a zero length payload it can never be fragmented so there is no need to be concerned about block replication flags.

If the querying application has any 'hints' that could indicate where the **GET** message might be forwarded to locate a response, the relevant locator(s) can be included in an optional JSON Metadata block. The items in the JSON would be:

http_auth A copy of the netloc field from the URI supplied by the application (if present). This will be assumed to be a netloc for an HTTP server that could be accessed over the HTTP CL if the query reaches a DTN<->HTTP gateway without the query finding a suitable response.

loclist An array of additional locators to which the query could be forwarded.

When a bundle with a BPQ *query* block is received at a DTN node, the *query string* is matched against the *query string* fields of any bundles in the (BPQ) cache in that node. If there is a match (in the case of NetInf **GET** requests any matching bundles would contain the same **URI** as in the query and hence would contain all or some fragment of the named NDO), a **GET-RESP** message *response* bundle is created either from the complete bundle or, if that is not present, from every matching fragment that coincides with the list of fragments requested (if specified) or from every fragment if there is no list. If the node is not the destination of the **GET** message and the whole NDO content was not sent as a response (whether in one or more fragments) then the **GET** message can be further forwarded, possibly with the requested fragment list modified as explained in Section 3.1.2. At the destination node for the **GET** message the bundle may be delivered to an application such as a gateway or router that may use a NRS or the supplied locators in the JSON Metadata block to forward the GET message to some other node that may be able to provide a response. The **GET query** bundle may request status reports (e.g., receipt or delivery reports) as with any BP bundle.

The format for a **GET-RESP** message depends on whether the server is returning both the content octets of the NDO and its affiliated data, or just the affiliated data if the server's cache does not contain the content octets and the request has not been propagated further. In all cases the response bundle will contain a *response* BPQ block (kind = 1) with the same *matching rule*, *query* and *query ID* fields as the *query* bundle that resulted in the response. If the response is only carrying the affiliated data, the affiliated data will be carried in a JSON Metadata block, the payload length will be zero and the bundle will also carry a payload placeholder Metadata block to fill in for the missing content. If both affiliated data and content are returned in the response, the content is carried as the bundle payload and the placeholder Metadata block is omitted. The BPQ block also carries the creation timestamp and EID of the original source of the bundle that carried the information included in the response³.

The **GET-RESP** bundle(s) are sent to the source of the **GET query** message that provoked the response(s). When the **GET-RESPONSE** bundles with BPQ *response* blocks are forwarded through DTN nodes on their way back to the *query* source, each node that has BPQ caching capabilities may decide to cache the response bundle so that it can service later **GET** messages requested the same NDO from its local cache.

Both the BPQ block and the JSON Metadata block should be marked for replication in all fragments in case the response is fragmented. This means that the fragments can be added to the BPQ cache in any nodes they traverse. Note that a response with a payload placeholder can never be fragmented because it has a zero length payload.

The affiliated data JSON object can contain the fields described below. These fields are identical to the ones that would be returned in the HTTP CL **GET-RESP**. Note that this remains a work-in-progress and is liable to change as the implementations mature and are further developed so the reader should expect a few minor inconsistencies between this description and the Internet-Draft and the code. As one might expect, the code is at the bleeding edge.

³ Note that the original source information only relates to the creation of the cached bundle in the DTN network. It may be that the actual content was created elsewhere and injected into the DTN node that published it into the DTN network. More information about the true origin of the NDO may be included in the affiliated data.

NetInf	String describing the version of NetInf protocol in use (e.g., "V0.2").
ni	The ni: scheme URI for the NDO retrieved.
msgid	Copy of the msg-id (BPQ <i>query</i> ID) from the GET message that resulted in the response.
ts	A timestamp. There are no fixed semantics for this as it is mainly for debugging, currently this contains the time at which the response was generated in most cases.
loc	A list of locator names from where the NDO might potentially be retrieved.
metadata	A JSON object containing content metadata recorded with the NDO when it was published plus information about how it was published.

Some implementations (including this one) also return additional JSON elements as listed below:

status	A code, taken from the HTTP 2xx success response codes indicating what has been returned.
ct	The MIME content type of the NDO content, if known.
searches	An array of JSON objects describing searches that have flagged up this NDO as matching the search criteria.

If the **GET** message is processed by an application at the destination EID, it may be forwarded to multiple destinations. Thus the application may receive multiple copies of the NDO from these destinations. If these are received in a timely fashion, the results can be combined before sending the **GET-RESP** message back to the source of the **GET** message in the DTN network. By definition, the content of the NDO received from any destination must be identical to that from any other as verified by the digest in the ni URI. Accordingly the content can be taken from any of these responses provided it can be verified against the ni URI digest. However, the metadata received from the various destinations may contain additional information.. It is possible to merge the responses to produce a single set of metadata to send back to the requester. This operation will generate an extended list of locators from which the NDO can be retrieved and possibly several sets of search tokens that were satisfied by the NDO during previous searches.

3.1.2.2 PUBLISH operation

The **PUBLISH** message is carried in a bundle with BPQ block, a JSON Metadata block carrying the affiliated data and, if the content is being published, the content is carried as the bundle payload. If the content is not being published, payload 'placeholder' Metadata extension block is added to the bundle and the payload is of zero length. The BPQ block is a *publish* block (kind = 3) and carries the two mandatory parameters for the **PUBLISH** message (*URI* and *msg-id*) which are the same as for the as the GET message (see Section 3.1.2.1). The following affiliated data is built into

ni	The ni: scheme URI for the NDO being published. (this is a duplicate of the BPQ <i>query string</i> but is useful when forwarding the PUBLISH message).
http_auth	A copy of the netloc field from the URI supplied by the application (if present). This will be assumed to be a netloc for an HTTP server that could be accessed over the HTTP CL if the query reaches a DTN<->HTTP gateway without the query finding a suitable response.
loclist	Locators where the NDO might potentially be retrieved (useful if the content is not being published here but the publisher knows where the content might be retrieved).
ct	The MIME content type of the NDO content (whether carried or not, if known).
fullPut	Boolean value indicating if the publication operation includes the content of the file or just metadata.
octets	If fullPut is true, this form field provides the content of the NDO and information about the type of the content.

- rform** Allows the user to choose a browser friendly Hypertext Markup Language (HTML)-encoded report on the publication or a JSON encoded report suitable for automated processing and/or sending through additional CL hops.
- ext** A JSON encoded object that can contain a *meta* field. The *meta* field is expected to contain a JSON object that is stored as metadata for the NDO content on the server whether or not the content itself is stored. In future, additional fields can be added to the ext object.

The *matching rule* in the *query* block is 0 (exact match required) so that when stored in the BPQ cache these bundles will be matched against **GET** requests.

The BPQ block and JSON Metadata blocks should have the block flag requesting replication in all fragments set so that the BPQ information and metadata would be carried along with all fragments if the content is fragmented. Note that a **PUBLISH** message without the content octets cannot be fragmented as it has a zero length payload.

If a DTN node receiving the **PUBLISH** message does not have this NDO stored already, and both policy and resources allow, the server will add the metadata and the content (if supplied) to its cache. Subsequent **PUBLISH** messages for the same NDO can be used to add the content if the first publication did not supply it or just to add additional or updated information to the affiliated data.

If the NDO or its metadata contained in a **PUBLISH** message is cached at a DTN node at which it is received, the publishing node can elect to be informed via BP status reports. The implementation of the CL described later in this document has introduced a new status report that explicitly notifies the publisher that a bundle has been published by being placed in the node's BPQ cache. This report may be standardized at some future time.

The affiliated data stored for an NDO records information about the NDO needed to fill in the JSON response object described in Section 3.1.2.1.

When the **PUBLISH** message arrives at the destination EID specified in the message, the bundle may be delivered to a gateway or forwarder application that has registered an appropriate service code with the DTN BPA on the node. This application may choose to forward the **PUBLISH** message to other destinations. Consequently the application may receive notifications that the NDO has been published on other nodes. The application can then build a **PUBLISH-RESP** message to send back to the sender of the DTN **PUBLISH** message. This message should consist of a bundle with a BPQ *publish* block copied from the original **PUBLISH** message but with the *matching rule* set to 2 (*never match*) so that no attempt is made to cache the bundle or match anything from the cache with it. The bundle should not have any Metadata blocks. The bundle payload contains a report of the **PUBLISH** activity carried out by the application.

It is possible that, as with **GET** messages, the application may receive multiple **PUBLISH-RESP** messages. Unlike the **GET** case, it is not appropriate to merge the body of the response messages. Instead the application should concatenate the responses, interleaving information about the destination node that published the NDO and sent the **PUBLISH-RESP**. If the response format requested is JSON, then the individual JSON results can be combined into a single JSON object with the individual responses keyed by the locator of the publishing node that generated the response. For other formats, the responses can be interspersed with suitable text such as "From <locator>:<newline>". The concatenated reports are returned as the payload of the **PUBLISH-RESP** bundle.

If a JSON format report is requested, the same structure as used to return the metadata with a **GET-RESP** should be used.

3.1.2.3 SEARCH operation

The **SEARCH** message is carried in a bundle with BPQ block, a zero length payload and an optional JSON Metadata block. The BPQ block is a *query* block (kind = 0) and carries the two mandatory parameters for the **GET** message:

- tokens** the search string for the search is carried in the *query string* field..

msg-id the message IDentifier (ID) (must be unique per CL hop and request/response pair) is carried in the *query ID* field.

The *matching rule* in the *query* block is 1 (token match required) so that responses will contain a token string that overlaps with the *tokens* string in the query..

Because the **SEARCH** message has a zero length payload it can never be fragmented so there is no need to be concerned about block replication flags.

If the searching application has any 'hints' that could indicate where the **SEARCH** message might be forwarded to locate a response, the relevant locator(s) can be included in an optional JSON Metadata block as a **loclist** field. The JSON may also contain optional fields **rform** and **ext** as used in the PUBLISH form (see Section 3.1.2.2).

When a **SEARCH** message arrives at the destination selected by the searching application, the bundle may be delivered to an application or gateway that offers search services either by running a local or remote search application or forwarding the **SEARCH** request to another node that performs searches. For example a local search might be carried using some framework such as Lucene⁴ or a remote search might be carried out by passing the *tokens* to the search API of Wikipedia⁵. Whether the destination node forwards the **SEARCH** bundle is determined by local policy – depending on the topological location and capabilities of the node it may or may not be appropriate to propagate the **SEARCH** request.

The **SEARCH-RESP** message is bundle with a BPQ *response* block (kind = 1) with the *matching rule*, *query string* and *query ID* fields copied from the BPQ block in the **SEARCH** message that resulted in the response and the creation information set to indicate where and when the search response was created. No Metadata blocks are required. The payload of the bundle describes the results of the search.

The format of the payload depends on the value of the *rform* parameter supplied in the Metadata of the **SEARCH** request. If this is 'json' the search results from each searching node consist of a JSON encoded string with fields *NetInf* and *ts* as described in Section 3.1.2.1 plus search specific fields:

search A JSON object describing the search containing the *tokens* used plus information about the location and search mechanism used.

results A JSON array of objects, one for each NDO that matched the search *tokens*, each containing at least an *ni* field with a value of the ni URI of an NDO available at the node that made the search.

If the response format is other than JSON, the result is a human-readable string that gives the ni URI of each NDO that matched the search tokens and any other useful information that would allow the user to select from the list of answers. The result should also contain summary information equivalent to the JSON-encoded response.

As with the **PUBLISH** operation (see Section 3.1.2.2), a gateway or forwarder may need to combine **SEARCH** responses from several nodes. If the response format requested is JSON, then the individual JSON results can be combined into a single JSON object with the individual responses keyed by the locator of the publishing node that generated the response. For other formats, the responses can be interspersed with suitable text such as “From <locator>:<newline>”. The concatenated reports are returned as the payload of the **SEARCH-RESP** bundle.

The **SEARCH-RESP** destination is set to the the source of the **SEARCH** request and the bundle is reinjected into the DTN network. As this bundle has a BPQ *response* block, any node which receives the bundle and has a BPQ cache is at liberty to save a copy of the bundle prior to forwarding

Subsequently, when a **SEARCH** request bundle is received at a DTN node that has a BPQ cache, the node examines its cache to determine if there are bundles that have a *tokens* field that overlaps with the tokens list in the **SEARCH** *query string* (i.e., there is at least one common token in the list). If there are any such

4 Apache Lucene search framework - [Lucene web site](#)

5 Wikipedia Open Search API - [Mediawiki API](#)

bundles they will be sent as **SEARCH-RESP** messages. It is a matter of local policy whether the **SEARCH** bundle is forwarded after checking the local cache, and this may depend on whether any and how good the matches found in the local cache are.

3.1.2.4 Selection of Destination EID for NetInf Bundles and DTN Routing

The NetInf protocol is intended to allow for searching and retrieval of NDOs without knowledge of where the information is located. Thus it is not desirable for an application using NetInf over DTN to have to give a specific destination for NetInf requests. It would therefore be desirable to be able to specify the destination address using a 'wildcard' EID combined with a service demultiplexer and have the DTN network forward the bundle to as many nodes as it can find that support BPQ caching.

One simple way of achieving this would be to use epidemic routing but that may be considered overly resource hungry. An alternative which is not currently implemented would be to implement either the situation in which a DTN node can support more than one local EID. One of these EIDs could be the primary EID and would be a singleton EID used as its administrative address; other addresses could be implemented as multipoint EIDs so that nodes using this EID effectively become a multicast group. A second alternative would be to extend the DTN discovery mechanisms to allow nodes to advertise which services are implemented on nodes so that wildcard EIDs can be matched on service discriminator rather than just netloc component. Both of these options would require work to specify and implement them.

In the meantime it is probably sufficient to send NetInf bundles to a well known 'gateway' node that has connectivity to the Internet and could either forward the bundle to nodes in the Internet using HTTP or redirect them to other nodes in the DTN network, at some cost in latency, if it is aware of relevant nodes. Also if a node is made aware of locators that are expected to have a particular NDO in their cache through a locator list (e.g., returned with a search result or as part of the metadata from a **GET** request that doesn't supply the NDO content), then the request can be sent directly to such a node.

Reverse routing just requires using the source EID of the request as the destination of the response.

4. NetInf Device Overview

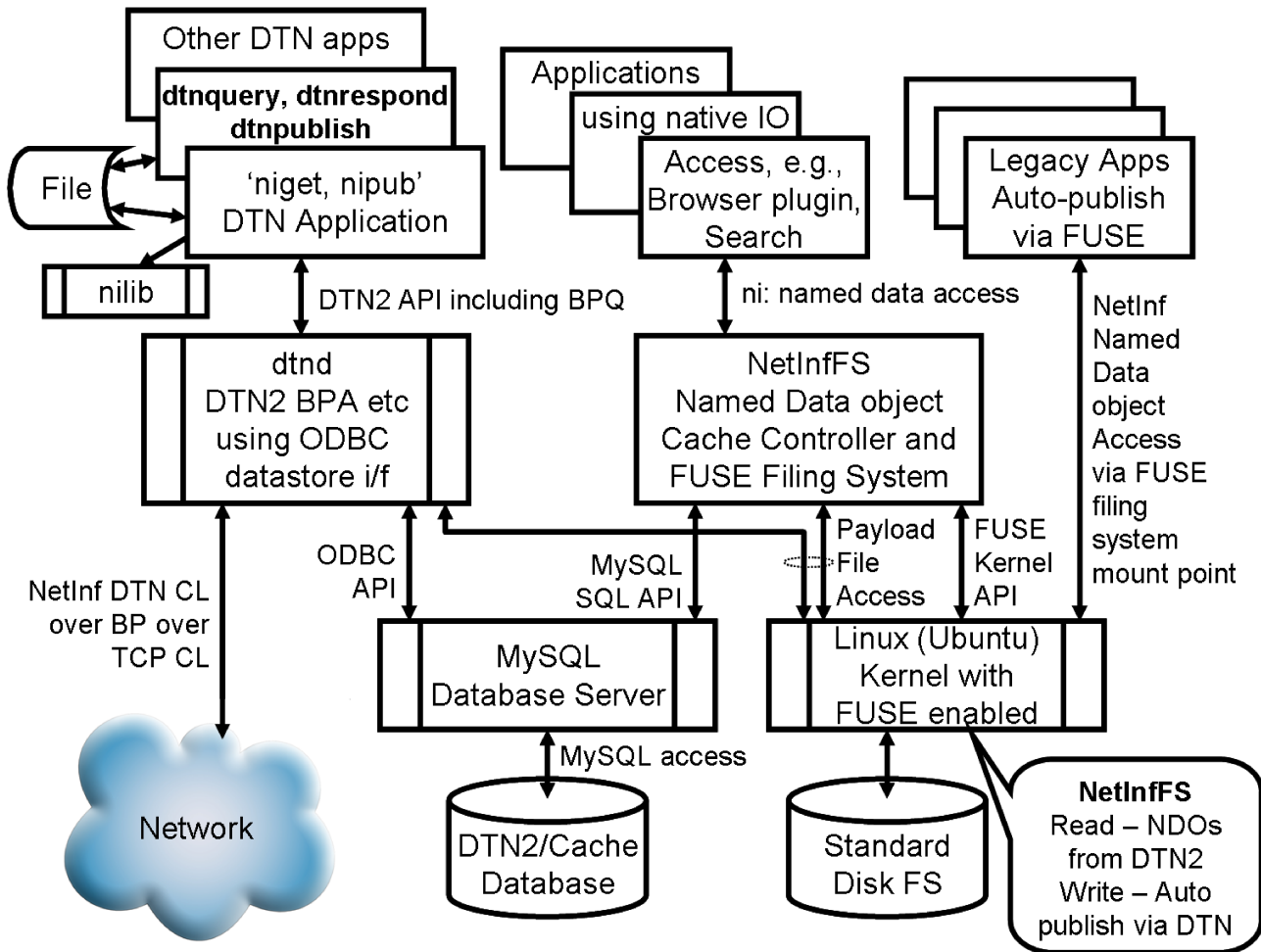


Figure 4: NetInf Device Architecture

Figure 4 provides an overview of the architecture of the *NetInf Device* that has been developed as part of TCD's work in the SAIL project as explained in Section 1.. The aims of the infrastructure components shown here are to

- ☑ enable 'conventional' application that generally access data via files and filing systems in a Linux environment to work with NDOs named with ni URIs without having to modify the data access interfaces in the applications, and
- ☑ demonstrate the use of the NetInf protocol over the DTN Convergence Layer.

In order to integrate the NetInf Device, which is intended to use the DTN CL, with the wider Internet, the DTN network needs to be linked to the Internet at the NetInf level by providing a gateway component. This was developed as part of a wider effort to implement a compatible NetInf Open Source Code Library (known as *Nilib* which provides support for several convergence layers including HTTP and DTN.

The overall network scenario in which the NetInf Device would be demonstrated is as shown in Figure 5.

Ultimately the Internet portion can be connected to the SAIL NetInf test bed network which has been established during the latter part of the project.

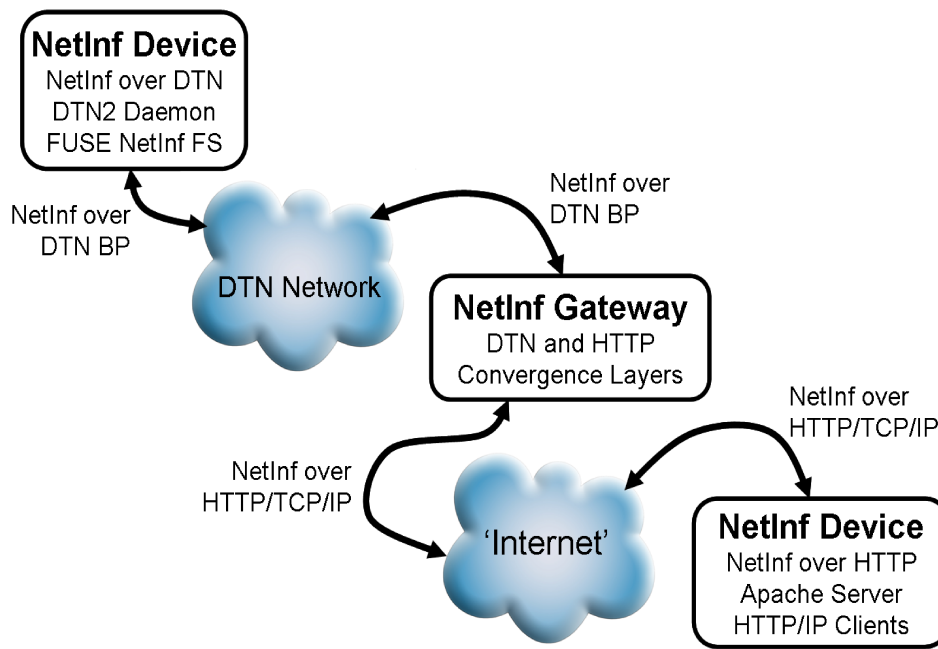


Figure 5: NetInf Device Network Context

It was decided from the outset to base the DTN work on the DTN2 Reference Implementation which has been maintained by TCD for a number of years under the auspices of two major EU Framework 7 programme projects, initially the Networking for Communication Challenged Communities (N4C) project and lately the SAIL project. Thanks are due to Alex McMahon who has managed the maintenance of the software for the last 4 years.

This work required design and development of four major components:

- Addition of BPQ extension block handling and management of the BPQ cache to the DTN2 core daemon (**dtnd**) to provide infrastructure for the NetInf message bundles as described in Section 3.1.2, together with various applications to allow injection of bundles such as NetInf messages with BPQ blocks and to implement a local searching mechanism using Lucene. This work is described in Section 5.
- Addition of a new persistent storage mechanism to DTN2 using the MySQL SQL database accessed via the ODBC (Open Data Base Connectivity) interface that allows external exposure of the bundle cache in DTN2 providing access to selected fields in the cached bundles via auxiliary storage tables and making it possible to link the bundle information to the payload files from outside the **dtnd** daemon. This work is described in a previous report [DTN2SQL]
- Creation of a FUSE-based pseudo-file system called **netinf** that allows the BPQ bundle cache to be viewed as files named by the ni URI names stored in the bundles' BPQ blocks and to write new files which are automatically published using an appropriate ni URI made by running a digest algorithm over the file content. This work is described in Section 7..
- Development of the Nilib NetInf code, as described in Section 6. to provide
 - the HTTP<->DTN CL gateway,
 - HTTP CL components to provide the NetInf protocol infrastructure in the Internet, and
 - the interface from the FUSE filing system to the automatic publication system using DTN2

It was decided that the NetInf Device would be implemented on a device running the Ubuntu distribution of the Linux operating system as DTN2 is well-established on this platform and it provides components that are needed to implement these components, including the FUSE filing system framework.

The code that has been written is available via the Sourceforge DTN and NetInf repositories as Open Source under the Apache 2 license and is freely available for experiment and usage.

Sections 5 to 7. give details of the work that has been carried out on the various software components to implement the functionality needed by the NetInf Device and to develop the Nilib Python code.

4.1 NetInf Device Operations

The DTN2 BPA daemon **dtnd** is built to support the BPQ extension block and ODBC-based persistent storage. The daemon is configured to use ODBC/MySQL persistent storage.

The **netinffs** is set up to access the same database as the DTN2 daemon is using for persistent storage and is informed of the directory where the DTN2 daemon stores the bundle payload files. When running the **netinffs** filing system provides a mount point that provides three directories:

- **bundles** which allows access to bundles by bundle number (read only)
- **ni** which allows access to bundles which have a BPQ block by the value of the *query string* in the BPQ block which will contain either ni URI name or a token list. (read only)
- **publish** which allows files to be written with arbitrary name and on closing, to be automatically published. The associated ni URI is available as an extended attribute once the file has been run through the digester used to create the ni URI digest. Once the file has been published the corresponding bundle will be visible in the **ni** directory and can be tied to the written file in the **publish** directory. If the file is updated then it will have to be republished with a new ni name corresponding to the revised content. It is intended that the sequence of ni names that represent various revisions of the file are maintained as an extended attribute. This effectively implements a form of version control system for the file and allows different versions to be retrieved subject to the bundles having a sufficiently long expiry period.

The **netinffs** periodically access the MySQL database auxiliary bundles table which has a record for each bundle currently in the daemon's bundle cache (whether or not they have a BPQ block). These are indexed by their locally unique bundle number. The bundle payload can then be accessed as a file in the **bundles** directory with the bundle number as file name.

If the bundle has a BPQ block the value of the *query string* field is used to allow the file to be accessed in the **ni** directory. Bundles with BPQ blocks will unconditionally remain in the BPA cache until they reach their expiry time (as opposed to others which may be deleted early if they are deemed to have been delivered or forwarded in such a way that they are not required by the BPA for further forwarding, etc. Of course, the time they remain in the cache is dependent on the expiry period specified when the bundle was created.

Combining the **netinffs** access with conventional DTN2 BPQ related applications (**dtnquery**, **dtnresponse**, and **dtnpublish**) and the Nilib NDO command line routines (including **nigetalt** and **nipubalt**) provide a variety of ways to access NDOs either from scripts or by conventional file access via existing applications.

Adding an Nilib HTTP<->DTN gateway to the device allows web access to NDOs via a conventional browser. The browser can send NetInf requests from a form provided by the gateway. The generated NetInf over HTTP requests are converted to NetInf over DTN requests by the gateway and sent out over the DTN network when a link is available. Depending on the latency of the DTN connectivity, the responses to these requests may be received before the web browser request times out and can be displayed directly. Otherwise they should be received some time later and will be placed in the gateway's cache which can be displayed by the browser at any time.

Search operations can either be carried out using the local Lucene-based search operation designed and implemented by Aidan Lynch as part of his Master's Thesis work [LYNCH2012] that uses the DTN BPQ blocks to carry the search request or by invoking a Wikipedia query from a NetInf-enabled HTTP server that has a connection to the Internet.

5. Additions and Modifications to DTN2 Code

The work on the DTN2 code base can be subdivided into a number of areas:

- 1 Introduction of ODBC-based SQL databases for persistent storage
- 2 The introduction of an auxiliary storage table for items of bundle data accessible by other programs including some items from BPQ blocks if they are included in bundles..
- 3 Bug fixes and extensions to the BPQ extension block processing and BPQ cache mechanism to support NetInf including support for PUBLISH operations and provision of field to carry the msg-id needed to link requests and responses. Rationalisation and refactoring of the code in BPQ supporting applications.
- 4 Introduction of 'publication' status reports to support NetInf NDO publication.
- 5 Bug fixes and improvements to the processing of Metadata blocks, especially across the DTN2 API.
- 6 Introduction of JSON ontology for Metadata blocks. Addition of ability to add JSON Metadata blocks to ICN bundles generated and received by ICN related DTN2 applications (**dtnpublish**, **dtnquery** and **dtnrespond**).
- 7 Completion of SWIG-generated scripting interfaces. I discovered that the existing interfaces could not handle the addition of either general extension blocks or metadata blocks both when sending and when receiving blocks from the network.
- 8 Bug fixes and validation work to ensure that the behind-the-scenes copy and destroy that results when the `push_back` and `append` methods of the `std::list` class are used to handle lists of blocks do not result in problems stemming from dynamically allocated strings in various extension block `BP_Local` structures.

The work described here, especially item 3, builds on the initial implementation of BPQ blocks and the BPQ cache within DTN2 carried out by Aidan Lynch as part of his Master's Thesis work during 2011 which was also partly supported by the SAIL project.

The various segments of work are described in more detail in the following subsections.

5.1 ODBC-based SQL Persistent Storage

This work has been described in a previous report [DTN2SQL] that built on the initial work to add SQL-based storage to DTN2 as well as other capabilities [SCOTT2011]. The DTN2 BPA (**dtnd**) program used by the NetInf Device is compiled with ODBC support (**configure --with-odbc**) and configured to use a MySQL database for persistent storage.

5.2 Support for Auxiliary Data Table for Bundles in MySQL Database

The enabling work for this capability has been described in a previous report [DTN2SQL]. The DTN2 BPA (**dtnd**) program used by the NetInf Device has BPQ block support compiled in (**configure --with-bpq**) and is configured to use a MySQL database for persistent storage. This makes certain of the fields in bundles in the DTN2 cache separately visible to external applications via the *bundles_aux* table in the database. The fields are:

DB Column Name	Content	Type
bundle_id	Bundle sequence number in this BPA (also used as key)	integer
creation_time	Creation time for bundle as seconds since the Epoch	big integer
creation_seq	Creation sequence number in creating BPA	big integer
source_eid	Endpoint Identifier of bundle source -	string
dest_eid	Endpoint Identifier of bundle destination	string
payload_file	File name for payload file	string
bundle_length	Overall length of bundle (total across fragments)	integer
fragment_offset	Offset of fragment payload in complete payload	integer

DB Column Name	Content	Type
fragment_length	Length of payload in this fragment (0 if not a fragment)	integer
bpq_kind*	Query(0), Response (1), Response – don't fragment (2) or Publish(3)	small integer
bpq_matching_rule*	Exact match (0), Token match (1) or Never match (2)	small integer
bpq_query*	Query string in BPQ block (ni URI or set of search tokens)	string
bpq_real_source*	Endpoint identifier of original source of response bundle	string

* These columns are only filled in when the bundle contains a BPQ block.

The DTN2 software is now constructed so that adding additional columns to the *bundles_aux* table is relatively simple. The following steps are needed:

- Select the field from the Bundle class to add to be added to the table.
- Determine the format of the database column that will be used to hold the data in *bundles_aux*. For strings of indeterminate length using VARCHAR(2000) is the suggested solution which allows for strings up to 2000 octets. The supported column types are defined by the *detail_kind_t* in **oasys/storage/StoreDetail.h** with the mapping to ODBC column data types defined in **oasys/storage/ODBCStore.cc**.
- Choose a column name for use in the *bundles_aux* table.
- Determine how to express the required column type in the SQL table declaration. Add the declaration to the **CREATE TABLE bundles_aux** in **DTN2/sqldefs/create_aux.sql**.
- Add a corresponding extra call to the *add_detail* method in the class constructor of **BundleDetail** in **DTN2/servlib/bundling/BundleDetail.cc**. The ordering of columns in the table and *add_detail* calls in the constructor are not related and order is not important for either of the lists. The *add_detail* method links the name of the database column and its SQL type to the data and its length in the specified bundle.
- Determine whether this field is in the core data of the Bundle or one of the extension blocks. For fields from the core data, the data can be transferred directly. Otherwise, it will be necessary to locate the required block and extract the data from it. The existing code shows examples for both cases.
- Initialize or tidy the database to recreate the *bundles_aux* table. The table schema can be inspected via the *isql* command (enter **help bundles_aux**) or *mysql* command (enter **describe bundles_aux;**)

5.3 BPQ Extension Block Processing and BPQ Cache

The initial implementation of the BPQ Extension Block processing and the BPQ cache in DTN2 was carried out by Aidan Lynch as work towards his Masters Degree in Computing at Trinity College Dublin. This work is documented in the dissertation that he submitted as part of his degree work [LYNCH2012].

Subsequently, a number of minor issues were discovered when initially testing this work for use with the NetInf Device. Also, for use with the NetInf protocol, it was found that an additional field named *query_id* was needed to carry the *msg-id* that is used to correlate NetInf responses with the original request. The addition of the *query_id* was required because the information placed in a response bundle by DTN2 does not contain any information that can provide this correlation; furthermore, the *msg-id* is needed when forwarding the NetInf request to other nodes whether using the DTN or HTTP CL.

Additionally certain fields from the BPQ extension block were to be placed into the *bundles_aux* database table as described in Section 5.2. This required additions to the **BundleDetail** class constructor and the SQL table creation script as explained in Section 5.2.

The opportunity was also taken to improve performance by creating the internal representation of the BPQ block data (as specified in the **BPQBlock** class – see **DTN2/servlib/bundling/BPQBlock.h**) once for all when the enclosing generic **Block** class instance is initialized. The **BPQBlock** class is derived from the

BP_Local class which allows a reference to the resulting **BPQBlock** class instance to be stored in the *locals_* private data of the Block instance and accessed whenever individual details of the BPQ block structure are needed rather than decoding the serialized form multiple times. In support of this, a copy constructor was added to the **BPQBlock** class declaration. This may be needed when BPQ extension blocks are inserted into one or other of the block lists used in the **Bundle** structure. These lists use the **BundleList** class which is derived from the C++ standard library template class **std::list**. If an insertion into such a list extends the reserved length of the list, the entire list is copied to a new data structure. The elements in the list have their copy constructors called 'under the covers' to do this and then the old copies are deleted. Apart from the potential performance penalties of this process, it is vital to ensure that any dynamically allocated memory is correctly handled by the copy constructor. The code now behaves correctly and in various places the size of the list being created is precalculated and the requisite number of entries reserved prior to starting to fill the list so that the list need not be resized during filling. In the case of decoding the bundle as it is received from the network when it is not possible to know the number of blocks in advance, the creation of the auxiliary data structures has been postponed until the final *validate* phase of processing incoming blocks by which time the lists are complete. This avoids the *locals_* structures having to be copied repeatedly as the list is extended.

The work on the BPQ extension block has been carried out in two phases and is incorporated into the DTN2 code base in several changesets. The work for each of these changesets is described separately.

5.3.1 BPQ Extension Block and BPQ Cache Changeset 3550

This changeset was registered at 2013/07/03. An extra application (*apps/dtnpublish*) has been written to allow bundles containing a Query Extension (BPQ) block of the (new) kind *KIND_PUBLISH* to be sent out into the network where they will be cached by appropriately configured nodes to which they are forwarded.

The following changes have been made:

- a new application '*apps/dtnpublish*' has been written. It has numerous options - see the **dtnpublish** manual page or use '*dtnpublish --help*'.
- Bundles that are in the BPQ cache will not be subject to early deletion so they will remain available in the cache until their lifetime expires or cache space runs out so they have to be deleted as the least recently used.
- Bundles of *KIND_PUBLISH* are cached in the node where the application API created the bundle with a Query Extension Block (as opposed to responses). The *KIND_RESPONSE* bundles were previously not cached because of the bug mentioned below which prevented the response bundles being returned correctly. This problem is now fixed, but at present *KIND_RESPONSE* blocks are NOT cached in the node where the bundle is originated. This is partly because there is a logical problem if the response was externally generated from an existing cached bundle. This situation may be altered in future.
- Responses derived from cached bundles that were cached with *KIND_PUBLISH* BPQBlocks, will have their responses mutated to *KIND_RESPONSE*.

In the process a number of bugs identified with the initial Query Extension block (BPQ) code (changeset 3528):

- Bundles have to be deleted from the appropriate BPQ cache entry when they expire or are otherwise deleted. If the cache entry is then empty it has to be deleted also.
- An apparently long standing bug was identified and fixed: bundles containing extension blocks and metadata blocks that were received from the API and delivered directly to an application on the same node would not have the extension blocks or metadata blocks attached to the delivered bundle. This was because **APIServer::dtn_rcv** method did not scan either the *api_blocks* or *generated_metadata* vectors when creating the structure to pass back to the application.

- The **BlockInfo::type** method now allows for all the different sorts of extension block that have been defined.

Some general improvements to the BPQ-related code have been made:

- To avoid repeated recreation of the internal representation of the data in the Query Extension Block, a **BPQBlock** instance is created when the **BlockInfo** instance representing the Query Extension Block is created. This occurs when:
 - The *dtm_send* API function is called with a BPQ extension block
 - A bundle is received from a link containing a BPQ extension block
 - A bundle is read back from persistent storage on bundle daemon startup
 - A response bundle is created as a result of receiving a BPQ QUERY

Note that it is not necessary to duplicate the **BPQBlock** into **BlockInfo** instances placed into *xmit_blocks* vectors created during bundle transmission as all the routines processing these blocks treat them as opaque data.

- To manage this data **BPQBlock** has been made a derived class of **BP_Local**. The **BlockInfo** class provides for the storage of a pointer to a **BP_Local** class instance which can be written and accessed by *set_locals()* and *locals()* methods respectively. A **BPQBlock** instance is created from the opaque data in the extension block when the **BlockInfo** is created. Classes that are designed to process BPQ blocks can then 'downcast' the **BP_Local** pointer type to the **BPQBlock** pointer type using *dynamic_cast*. This capability existed in DTN2 but was mainly used in the Bundle Security protocol code.
- As part of the above change, the original creation timestamp and source EID have to be written into the opaque data (and the **BPQBlock**) when a bundle with a Query Extension Block is received in the **APIServer::dtm_send** method. This has been implemented by adding an extra **Bundle** pointer parameter to the **BlockProcessor::init_block** method and making this method virtual so it can be overloaded in other types of **BlockProcessor**. The extra parameter is ignored by the generic method in the base class (**BlockProcessor**) but, if it is not NULL, it may be used by the overridden method in **BPQBlockProcessor** to set the creation timestamp and original source EID. In the other cases where new **BlockInfo** structures are created for Query Extension blocks, the opaque data used will already contain the correct timestamp and source EID so the **Bundle** pointer parameter is left as NULL which signals the *init_block* routine to leave the opaque data unchanged. [Note this mechanism has been further altered in subsequent changes.]
- In association with this the initialization methods for **BPQBlock** have been restructured so that a **BPQBlock** instance can be created directly from the opaque data in the **BlockInfo** instance rather than requiring the data in the bundle each time. It is also no longer necessary to differentiate between locally and remotely created **BPQBlocks**.
- Similarly, a specialized **BPQBlockProcessor::reload_post_process** method has been added to create the **BPQBlock** instance and save it in the **BlockInfo** instance when pulling bundles in from persistent storage.
- The **BPQBlockProcessor::consume** method now creates the **BPQBlock** and records it in the **BlockInfo** instance when the block is complete. [Again this change has been modified to postpone the creation of the **BPQBlock** to the **BPQBlockProcessor::validate** method.]
- New setters have been added to update the *creation_ts* and *original_source* fields in **BPQBlock** and a *validated_flag* has been added to record if the block is correctly populated.
- **BPQBlockProcessor::generate** has been modified to make better use of the 'source' block set up by **BPQBlockProcessor::prepare** rather than creating information afresh.

To assist with management and monitoring, some additions and improvements have been made to the BPQ commands and the bundle info command:

- added command **bpq list** which lists the query values for the current BPQ cache entries
- added command **bpq lru** which lists the query values in the BPQ cache in the order of the least recently used (LRU) list with the most recently used item first.
- improved **bundle info <bundleid>** command so that it prints the contents of the BPQ block as well as its existence and size. This is done by providing an overloaded **BPQBlockProcessor::format** method rather than using the base class version.
- Additional locks have been provided to ensure that the whole BPQ cache is locked when manipulating the list of cache entries as well as the existing locks for each fragment list.

5.3.2 Changes to BPQ Extension Block and BPQ Cache

As of the publication of this report this changeset is in preparation and will be submitted for incorporation into the main source code very shortly. The main purpose of these changes is to add the *query_id* field to the **BPQBlock** in order to carry the *msg-id* used by the NetInf protocol to correlate requests and corresponding responses. Some additional work has been carried out to manage the association between the on-the-wire binary representation and the internal **BPQBlock** structure referenced through the *locals_* element in the **Block**.

- The use of **BPQBlocks** in both the API and on the generation of responses to queries internally in the BPA requires elements of the **BPQBlock** to be initialized from multiple different sources. The **BPQBlockProcessor::init_block** method is used to carry out this initialization. This method is a virtual method that is declared and implemented in the base class **BlockProcessor** from which the various block-specific block processing classes, including **BPQBlockProcessor** are derived. In order to localize the code that modifies the internal data of the **BPQBlock**, a generic mechanism has been created that allows a set of flags to be passed in to the **init_block** method which can be interpreted by specific specializations of the **init_block** method. The two cases where fields of **BPQBlock** are modified are
 - On creation of a bundle with a **BPQBlock** from the **APIServer::handle_send** method, the initial values of the creation timestamp fields have to be copied from the main bundle metadata (as described in Section 5.3.1). In this case all the remaining fields are initialized from the data supplied by the application in the API call but the application is not able to correctly set the creation data.
 - On creation of a response bundle from a bundle in the BPQ cache, the *query_id* field has to be copied from the **BPQBlock** in the bundle that contained the query instead of the value in the cached bundle. In this case the creation timestamp fields and all the other fields are copied from the cached bundle.

To avoid polluting the base class with derived class specific items and to allow elements (such as the **BPQBlock**) to be omitted from instantiations of the BPA by appropriate configuration, the concept of *transfer_flags* has been introduced. A class variable *transfer_flags_* in **BlockProcessor** holds the next available flag bit. Derived processors can allocate a new flag bit on first use by calling **BlockProcessor::new_transfer_flag**. A parameter has been added to **BlockProcessor::init_block** and all the corresponding routines in derived classes to pass in relevant transfer flags. Flags for the two cases mentioned have been created and are used as appropriate in the calls of **init_block** in **DTN2/applib/APIServer.cc**, **DTN2/servlib/bundling/BPQBlockProcessor.cc** and **DTN2/servlib/bundling/BPQResponse.cc**. Note that the avoidance of pollution was not total – because of calling **init_block** for all blocks in **APIServer.cc** the *create_ts* flag had to be created in **BlockProcessor** rather than **BPQBlockProcessor**.

- When the **BPQBlock** was introduced (changeset 3528) the file **DTN2/applib/dtn_types.h** was manually modified. This is not supposed to happen because this file is automatically generated by *rpcgen* from **DTN2/applib/dtn_types.x**. The BPQ specific types that were added are not needed by the generated RPC

code for the API. They have now been moved to a separate header file **DTN2/applib/bpq_api.h** so that **dtm_types.h** can be generated from **dtm_types.x** again. Note that some changes have been made to **dtm_types.x** (see next bullet and Section 5.4).

- For convenience in examining received bundles, the field *payload_len* has been added to *dtm_bundle_payload_t* in the API specification file **dtm_types.x**. This field is not checked or used when sending bundles from applications but is set when a bundle is retrieved using receive or peek bundle.
- The field *query_id* and the associated *query_id_len* have been added to the **BPQBlock** specification. *query_id_len* is encoded as an SDNV and represents the length in octets of the string in the *query_id* field. The *query_id* is intended to carry a random string that can be used to correlate requests and responses as in the NetInf protocol. To assist with generating suitable strings, the function *make_random_string* has been provided. The code is in **DTN2/applib/mkrandstr.c** and a corresponding header file **mkrandstr.h** is also available.
- The three applications provided in DTN2 that handle BPQ blocks (*dtmquery*, *dtmresponse* and *dtmpublish*) have been refactored and some common code transferred to **DTN2/applib/bpq_api.c**. The interfaces for these common parts are declared in **bpq_api.h** along with the structure definitions removed from **dtm_types.h**.
- An enumerated type definition for the *matching_rule* field in the BPQ block has been added to **bpq_api.h**. Code that reference the matching rule in **BPQBlock** has been altered to use the type *matching_t* in place of *u_int* as appropriate.
- Constant definitions for the ontology values of the two metadata block kinds used in NetInf have been added to **bpq_api.h**.
- An enumerated type providing values for the type field in different sorts of BP blocks has been added to **DTN2/applib/dtm_api.h** mirroring the corresponding BPA table in **BundleProtocol.h**.
- An enumerated type providing values for the ontology specifier in metadata blocks has been added to **dtm_api.h**. This provides values for the one formally defined ontology (*ONTOLOGY_URI* = 1) and the lower and upper limits for experimental ontologies (respectively 192 and 255).
- Support for adding JSON encoded metadata to NetInf protocol bundles has been added to the applications that handle BPQ blocks (*dtmquery*, *dtmresponse* and *dtmpublish*). Applications that send bundles now have an option for inserting a JSON ontology Metadata block (see Section 5.6). To support this a simple JSON checking parser and pretty printer has been added to the code; the code is in **DTN2/applib/json.c** with associated declarations in **json.h**. The JSON code does not cater for modifying the generated JSON parse tree. It is expected that the metadata value will be supplied as a JSON encoded string that can be checked using the parser and may be displayed with the pretty printer.
- Command line options were added to *dtmpublish* and *dtmquery* to allow the user to specify the metadata string and the *query_id* (aka the *msg-id*).
- The makefile for the applications library (**DTN2/applib/Makefile**) has been modified to compile the three extra components (**bpq_api.c**, **mkrandstr.c** and **json.c**).

5.4 Introduction of 'Publication' Status Reports

When a bundle with a **BPQBlock** with a *bpq_kind* of any of *BPQ_BLOCK_KIND_RESPONSE*, *...RESPONSE_DO_NOT_CACHE_FRAG* or *...PUBLISH* arrives and the node decides to cache it in the BPQ cache, then it is desirable to inform the publishing node that bundle has been cached and can therefore be retrieved from the node that cached it. To do this, a new type of status report has been implemented. This is known as a 'publication receipt'.

Adding the status report is simple but seems to affect an inordinate number of files! These are in **DTN2/servlib/bundling** unless otherwise specified.

- The flag *publication_rcpt_* is added to the bundle metadata in **Bundle.h**.
- Accessor and setter routines for *publication_rcpt_* are defined in **Bundle.h**.
- In **BundleStatusReport.h** the flag *STATUS_UNUSED* (value 0x40) has been redefined as *STATUS_PUBLISHED* and an extra timestamp *publication_tv_* added to the *struct data_t*.
- The method **parse_status_report** in **BundleStatusReport.cc** has an extra segment added to encode the *publication_tv_* when the report flags indicate a publication report.
- In **DTN2/applib/dtn_types.x** an extra delivery option flag *DOPTS_PUBLICATION_RCPT* (value 1024) is added to *dtn_bundle_delivery_opts_t* and the field *publication_ts* is added to *dtn_bundle_status_report_t*.
- In **DTN2/applib/dtn_api_wrap.cc** added the fields *publication_ts_seqno* and *publication_ts_secs* to the *dtn_status_report* structure. In the functions **dtn_rcv** and **dtn_peek** copy the values for these two fields from the corresponding fields in the API *dtn_bundle_status_report_t* data structure when the bundle contains a status report.
- In **DTN2/applib/APIServer.cc**, the methods **handle_rcv** and **handle_peek** set the delivery option *DOPTS_PUBLICATION_RCPT* bit in the returned bundle specification data structure if selected in the bundle metadata and transfer the publication time in the status report structure if the bundle is a status report.
- Similarly, **handle_send** sets the *publication_rcpt* flag in the created bundle if the APR delivery option specify *DOPTS_PUBLICATION_RCPT*. (Applications can't generate status reports).
- A number of applications report on status blocks received and were extended to provide information for the publication receipt. In each case this involved duplicating the code for other status report topics:
 - **DTN2/apps/dtnperf/dtnperf-client.c**
 - **DTN2/apps/dtnping/dtnreporter.c**
 - **DTN2/apps/dtnping/dtntraceroute.c**
 - **DTN2/applib/bpq_api.c**
- In the process of inserting code for publication reports it was discovered that some of the code had not been updated to include some of the other delivery options. This was corrected in parallel with adding the publication report code:
 - In **APIServer::handle_send** testing if the bundle was asking reports with a source EID of *dtn://none* omitted *DOPTS_ACKED_BY_APP_RCPT*. Also the options flags *DOPTS_ACKED_BY_APP_RCPT* and *DOPTS_DO_NOT_FRAGMENT* were not transferred to the bundle metadata.
 - In **APIServer::handle_rcv** and **::handle_peek** the status report request *DOPTS_ACKED_BY_APP_RCPT* was not transferred from the bundle to the API data structure
 - In **DTN2/applib/dtn_types.x**, *DOPTS_ACKED_BY_APP_RCPT* was not defined..
 - In **DTN2/apps/dtnping** displaying status reports for *DOPTS_ACKED_BY_APP_RCPT* was omitted in both **dtnreporter.c** and **dtntraceroute.c**. In **dtntraceroute.c** a status report for *DOPTS_ACKED_BY_APP_RCPT* was also requested.

5.5 Metadata Block Processing

The NetInf over DTN CL described in Section 3.1.2 uses a Metadata extension block carry the affiliated data for NDOs being retrieved or published⁶. Accordingly it was necessary to ensure that DTN2 worked correctly

⁶ Reminder: the term affiliated data is defined in the NetInf Protocol specification [KUTSCHER2012] and subsumes metadata that is associated with the nature of the NDO (for example its content type, descriptions of the data and

with the current definition of Metadata Extension blocks [RFC6258]. It appears that the Metadata block subsystem in DTN2 has not been much exercised. A brief query to the DTN Users mailing list did not bring any users out of the shadows, so it seems that the problems noted below are long standing. I have attempted to remedy them but the solutions implemented in some cases might need further refinement.

5.5.1 Problems Detected

This section provides some more details on the problems detected in the handling of Metadata blocks.

5.5.1.1 Non-conformance with Published Specification

On inspection it became clear that DTN2 had not been synchronized with the public specification of the Metadata block. The block specific data of the Metadata block contains two fields:

- The *Metadata Type* (also known as the *Ontology*) encoded as an SDNV (Self-Defining Numeric Value)
- The *Metadata Field* that is an array of octets occupying all the remaining space in the block

The specification assumes that the length of the *Metadata Field* is defined by the overall length of the block data in the block header field less the (self-defining) length of the *Ontology* field. The code in DTN2 expected an additional SDNV length field after the *Ontology*. There were comments to the effect that this was incorrect but the code had not been adjusted.

5.5.1.2 Possible Inappropriate Use of API Data Structure

The API data structure in which Metadata blocks were exchanged between the BPA and applications was (arguably) being misused in that the extension block *type* field in the API structure *dtm_extension_block_t* (see **DTN2/applib/dtm_api.h**) was being used to carry the Metadata *Ontology* type rather than the extension block type rather than it being encoded as an SDNV in the *data_val* field.

5.5.1.3 Differences in Handling Between Generated and Received Blocks

The design of the DTN2 BPA follows the Metadata block specification that allows for Metadata blocks to be generated internally and added to a bundle before it is forwarded on a particular link. As far as I am aware, this capability has never been used but the BPA could handle it if required and there is extensive code to handle these blocks. There was a distinction between such generated blocks and ones read from the network in the way that the internal pointers to the *Metadata Field* data were handled. In the case of generated blocks it was expected that memory would be specially allocated for the *Metadata Field* content and explicitly deleted when the **MetadataBlock** instance was deleted. However for received blocks, the *Metadata Field* pointer was pointed into the received data string and memory was not allocated separately. This optimisation unfortunately failed to take account of the **std::list** copy/destruct mechanism discussed in Section 5.3. The consequence of this was that for non-generated bundles the *Metadata Field* pointer could become corrupt because it was just copied by the copy constructor but the underlying data got reallocated by the copy constructor with the result that the *Metadata Field* was pointing at freed memory.

5.5.1.4 Disposition of Metadata Blocks Received over Network and API

Looking at the code in **DTN2/applib/APIServer.cc::handle_send** before this work started (i.e., previous to changeset 3518), it appears that Metadata blocks attached to a bundle received from the API were to be treated as generated Metadata blocks (see Section 5.5.1.3) rather than an integral part of the bundle created by the registered application. This treatment does not seem appropriate as the metadata was created to go with the bundle and it isn't really different from data received from a network connection.

This also led to the same problem that was identified with other extension blocks whereby they were not incorporated into locally delivered bundles. A partial fix of this problem was added at changeset 3518 but this did not set up the *locals* pointer to the decoded data in the block. It also led to two copies of the block

search tokens that selected the data) and any data that is related to the operation of the NetInf protocol such as additional locators where it might be retrieved,

being added to bundles that were sent over network links rather than locally delivered. The second copy came from the insertion of the Metadata block into the *generated_metadata* list for the 'null link'. The *generated_metadata* structure potentially holds sets of extra generated metadata to be associated with particular links. This would be created to be sent across either a specific link or (using the 'null link' fudge) all links being incorporated into copies of the bundle sent on the specified link(s). This capability has not been much used. There was a proposal for modifying metadata blocks in transit in the Intentional Naming proposal [draft-pbasu-dtnrg-naming] (now long expired); the security protocol which was seen as a candidate for using this capability addresses the problem in its own way and doesn't use standard Metadata blocks. Any use of the generated Metadata blocks would almost certainly require the BPA to understand the semantics of the relevant metadata blocks: whether this is desirable is unclear but it might be acceptable if the metadata was associated with a routing protocol rather than being application-specific.

Turning to Metadata blocks received over network links, currently these blocks are placed on the *recv_blocks* list of the bundle and generally treated as opaque data. The Metadata block is also placed on the *recv_metadata* list of the bundle by **MetadataBlockProcessor::parse_metadata** but this list is currently 'write-only' within the BPA although it is made available to external routers and convergence layers. The *recv_metadata* list plays no part in generating the outgoing bundle data when the bundle is forwarded. However there is extensive code to allow received metadata to be modified before being added to outgoing bundles. As noted previously this capability is essentially unused.

5.5.2 Improvements and Fixes for Metadata Block Code

This section gives an overview of the remedies that have been adopted to remedy the issues noted in Section 5.5.1.

5.5.2.1 Carrying Metadata Blocks Across Application Programming Interface

On the BPA side, the methods **handle_send**, **handle_recv** and **handle_peek** in **DTN2/applib/APIServer.cc** that deal with the *dtm_extension_block_t* type and Metadata blocks have been modified to use the type in the same way as for other extension blocks, i.e., the block type is set to *METADATA_BLOCK* and the ontology and metadata field are carried in the data area of the *dtm_extension_block_t*.

Additionally, checks are performed to verify that the block types in the extension blocks and metadata list have appropriate block types.

On the application side, the following applications in the **DTN2/apps** directories that handle metadata blocks have been modified to match with the BPA side:

- **dtmpeek**
- **dtmrecv**
- **dtmsend**
- **dtmsource**
- **dtmublish**
- **dtmquery**
- **dtmrespond**

On reflection, this change may be inappropriate as it requires additional work in all the applications to encode the ontology as an SDNV. Since the problems with the format of the Metadata block on the wire have not caused issues, it perhaps indicates that the Metadata part of the API is currently little used but it would reduce the complexity of applications slightly if the 'misuse' of the *dtm_extension_block_t* was perpetuated. Reversing this change will be considered before the change is published.

5.5.2.2 Matching MetadataBlock Coding to Specification

The encoding of the 'on-the-wire' format of the Metadata block has been modified to conform to the specification in [RFC6258]. This affects the methods **handle_send**, **handle_recv** and **handle_peek** in **DTN2/applib/APIServer.cc** and **MetadataBlockProcessor::parse_metadata** in **DTN2/servlib/bundling**.

In order to avoid confusion with the overall DTN (extension) block *type*, the Metadata type is now known throughout as *ontology*.

Also checks have been inserted to check that the ontology value is a legitimate known value, currently either **ONTOLOGY_URI** (1) or in the agreed experimental range (192-255).

5.5.2.3 Dynamic Memory Allocation in Metadata Blocks

To avoid problems with hidden copying of Metadata blocks referenced through the *locals_* data member of the **Block** class that holds a **MetadataBlock**, all **MetadataBlocks** now allocate dynamic memory to hold the metadata field rather than pointing into the binary representation held in the **Block** class instance. A copy constructor that handles allocating this memory and copying the value from original to copy has been created. Accordingly there is now no need to distinguish generated and received blocks when creating, updating, copying or deleting this data.

5.5.2.4 Placement of Metadata Blocks when using API to Send Metadata

In **APIServer::handle_send** metadata blocks are now only placed in the *api_blocks* list of the bundle under construction. They are no longer placed in the 'null link' *generated_metadata* list. For convenience the method **MetadataBlockProcessor::init_block** has been created for consistency with other types of block and is used to construct the **Block** instance and the **MetadataBlock** referenced in its *locals_*. This method uses **MetadataBlockProcessor::parse_metadata** to create the **MetadataBlock** so that all metadata blocks are now created by a single method.

5.6 Introduction of JSON Ontology Type for Metadata Blocks

To support the NetInf protocol by carrying the affiliated data in various messages, a Metadata block ontology was defined that carried a JSON encode string as its metadata field. Initially this uses an ontology code from the experimental range but may be formally specified in future and a ontology code allocated from the main range.

Support for JSON strings in the API was introduced as described in Section 5.3.2.

5.7 Completion of SWIG Generated Scripting Interfaces

The NetInf NLib components that are used to interface with the DTN2 BPA are primarily written in Python. It was therefore decided to use Python scripting interface to send and receive bundles via the DTN2 API. Unfortunately when initial tests with the code were carried out it was discovered that the DTN2 scripting interface was incomplete. In particular it could not handle either the lists of extension blocks or lists of metadata blocks that are passed across the API if extension or metadata blocks are to be included in bundles.

Initial investigation seemed to indicate that the reason that this 'hole' had been left in the scripting interface was that the SWIG tool⁷ was not able to deal generically with nested structures. The arrays of extension block structures in the bundle interface appears to be just such a nested structure.

At first it appeared that this may prove to a 'showstopper' since it might just be possible to write scripting language specific code for the Python interface but this would definitely not be generic and it looked as if it

⁷ SWIG (Simplified Wrapper and Interface Generator) is an open source software tool that converts an interface wrapper program, written in either C or C++, and an interface specification file into an interface module in one of a variety of scripting languages. DTN2 uses SWIG to provide scripting language support for its API in PERL, TCL and Python (version 2).. More details on SWIG can be found at [SWIG web site](http://www.swig.org).

would be very complex. Investigating the problem on the web seemed to confirm that there was no easy solution, at least in the general case.

I was just about to give up when I noticed that in fact somebody had done all the hard work for me. Essentially it was a solution for the particular case that we have for DTN2 rather than a fully generic solution to the nested structure problem, but that is of no consequence. The SWIG library turns out to support vectors of structures that are derived from the STL/C++ library vector template. Since the arrays of extension and metadata blocks can be expressed as vectors of extension block objects, this provides all the support that is needed. This useful library is documented in the SWIG manual at [SWIG Library - STL/C++ for vectors](#) (library section 8.4.2). I was somewhat surprised that this capability had not been mentioned in any of the many discussions of essentially similar problems on forum postings. Thanks are due to a certain Luigi "The Amazing" Ballabio who is credited with writing the SWIG **vector.i** code without which all this would have not been possible (at least in a finite time).

After some experimentation, I was able to build the requisite *dtm_extension_block_list* structure and provide a constructor for a specific length of list and a destructor that are 'attached' to and extend the standard mechanisms that SWIG constructs for a C structure. This effectively turns what is a C structure into a class. See section 5.5.6 of the SWIG manual [Adding Member Functions to C Structures](#). This is slightly odd because the wrapped interface is a C++ interface. However the 'hybrid' appears to work at least in Python. The relevant slightly magic incantations to provide the vector structure are added to **DTN2/applib/dtm_api.i** this is then used in the SWIG interface wrapper **DTN2/applib/dtm_api_wrap.cc** in a conventional way to move data from the wrapper data structures to the standard API structures (in **dtm_send**) and vice versa (in **dtm_recv** and **dtm_peek**).

Two extra parameters have been added to the **dtm_send** method to carry the extension blocks and metadata blocks methods. The *dtm_extension_block_list* structure holding the extension blocks is shown in Figure 6.

```
struct dtm_extension_block {
    unsigned int type;
    unsigned int flags;
    string      data;
};

typedef struct dtm_extension_block *dtm_ext_block_ptr;
struct dtm_extension_block_list {
    vector<dtm_ext_block_ptr> *blocks;
};
```

Figure 6: Structures used to carry extension block lists in scripting interface

The *dtm_bundle* structure returned by **dtm_recv** and **dtm_peek** has been extended to carry the lists of extension blocks and metadata blocks as shown in Figure 7.

```
struct dtm_bundle {  
    string      source;  
    string      dest;  
    string      replyto;  
    unsigned int priority;  
    unsigned int dopts;  
    unsigned int expiration;  
    unsigned int creation_secs;  
    unsigned int creation_seqno;  
    unsigned int delivery_regid;  
    string      sequence_id;  
    string      obsoletes_id;  
    string      payload;  
    unsigned int payload_len;  
    bool        payload_file;  
    unsigned int extension_cnt;  
    unsigned int metadata_cnt;  
    vector<dtm_ext_block_ptr> *extension_blks;  
    vector<dtm_ext_block_ptr> *metadata_blks;  
    dtm_status_report* status_report;  
};
```

Figure 7: Structure of a bundle with extension block lists in scripting interface

For convenience, the lengths of the two lists are carried as separate entries *extension_cnt* and *metadata_cnt*.

The text box (Figure 8) shows an example of how to build a list of metadata blocks to add to a **dtm_send** call in Python:

```
# Build an extension blocks structure to hold the maximum of 2 blocks
meta_blocks = dtnapi.dtn_extension_block_list(2)

# Construct the JSON metadata block (if any)
if req.metadata is not None:
    md = Metadata()
    md.set_ontology(Metadata.ONTOLGY_JSON)
    md.set_ontology_data(json.dumps(req.metadata.summary("http://example.com")))
    json_block = dtnapi.dtn_extension_block()
    json_block.type = METADATA_BLOCK
    json_block.flags = 0
    json_block.data = md.build_for_net()
    meta_blocks.blocks.append(json_block)

# Construct a payload placeholder for GET case with no content
# This distinguishes an empty payload file from the no content case
if (req.make_response and
    (req.req_type == HTTPRequest.HTTP_GET) and
    (req.content == None)):
    md = Metadata()
    md.set_ontology(Metadata.ONTOLGY_PAYLOAD_PLACEHOLDER)
    md.set_ontology_data("No content available")
    pp_block = dtnapi.dtn_extension_block()
    pp_block.type = METADATA_BLOCK
    pp_block.flags = 0
    pp_block.data = md.build_for_net()
    meta_blocks.blocks.append(pp_block)

...

# Send the bundle
bundle_id = dtnapi.dtm_send(dtm_handle, regid, src_eid, destn_eid,
                           report_eid, pri, dopts, exp, pt,
                           pv, ext_blocks, meta_blocks, "", "")
```

Figure 8: Sending a bundle with metadata blocks using Python interface

The second text box (Figure 9) shows how to examine the metadata blocks in a received bundle:

```
# Get the received bundle
bpq_bundle = dtnapi.dtn_rcv(dtn_handle, dtnapi.DTN_PAYLOAD_FILE,
                             rcv_timeout)
...

# Does the bundle have a Metadata block of type JSON or
# PAYLOAD_PLACEHOLDER
json_data = None
has_payload_placeholder = False
if bpq_bundle.metadata_cnt > 0:
    self.logdebug("Metadata count for bundle is %d" %
                  bpq_bundle.metadata_cnt)
    for blk in bpq_bundle.metadata_blks:
        if blk.type == METADATA_BLOCK:
            md = Metadata()
            if not md.init_from_net(blk.data):
                self.loginf("Bad Metadata block received")
                break
            if md.ontology == Metadata.ontology_JSON:
                json_data = md
            elif md.ontology == Metadata.ontology_PAYLOAD_PLACEHOLDER:
                has_payload_placeholder = True
                self.logdebug("Have placeholder: %s" %
                              md.ontology_data)
            else:
                self.logwarn("Metadata (type %d) block not processed" %
                              md.ontology)
```

Figure 9: Examining the metadata blocks in a bundle using the Python interface

5.8 Preallocation of BlockInfo Lists and Creation of BP_Local Data

The class **BlockInfoVec** is extensively used in DTN2 bundles to hold lists of the blocks from which bundles are constructed. **BlockInfoVec** has the STL/C++ standard library templated class **std::vector** as a base class. As lists are assembled a number of the methods such as *append* and *push_back* of the **std::vector** base class are invoked to insert blocks into the list. These methods can be rather inefficient in that if they need to expand the underlying array of items in the vector, this is done by reallocating the vector and *copying* (rather than moving) all the existing items into the new vector. The old entries are then deleted. The copying and deletion are all hidden inside the **std::vector** implementation.

This means that care has to be taken that there is a copy constructor for any item held either directly in a vector or indirectly through being a data member of an item held in a list.

In order to avoid repeated copying and deletion, it is possible to preallocate the underlying array to a size that will hold all the entries expected to be placed in the list using the *reserve* method. This has now been done in several places in the DTN2 code. These include:

- DTN2/applib/APIServer.cc method *dtn_send*.
- DTN2/applib/dtn_api.i in the construction of scripting interface extension block lists.
- DTN2/applib/dtn_api_wrap.cc in the construction of bundle extension and metadata block lists in *dtn_rcv* and *dtn_peek*.
- DTN2/servlib/bundling/BPQResponse.cc in *create_bpq_response* when copying the *api_blocks* and *rcv_blocks* vectors.

However, when bundles are being constructed from data received over the network, the BP does not provide an indication of the number of blocks making up a bundle in the primary block so it is impossible to effectively preallocate the block vector. To minimize the amount of copying and reallocation in this case, the

construction of the internal format of the block data stored in the *locals_* data member should be postponed until the vector has been completed. This can be handled by generating the internal form in the *validate* method for the block type which is called once the complete bundle has been received. This has been done for the **BPQBlock** class which has an extensive local data structure (see **BPQBlockProcessor::validate**).⁸

Similarly, the generation of the internal form can be handled in the *reload_post_process* method when the blocks are being read in from persistent storage when the BPA is being restarted.

6. Development of Nlib Python Code

An extensive library of code has been created in the Python scripting language to implement the NetInf Information Centric Networking protocol running over HTTP/TCP and DTN Bundle Protocol convergence layers. This library has been developed primarily by Elwyn Davies with contributions from Stephen Farrell.

The two convergence layers interoperate via a gateway that can receive messages over either convergence layer and forward them over the same or an alternative convergence layer depending on the availability of transport protocols on the link to the next hop NetInf node.

6.1 A Little History

During the earlier stages of the SAIL project, a number of participants cooperated in developing an information centric naming scheme that would use a cryptographic hash digest of the information object to tie the name to the content in a way that could not be readily forged and could be verified by a user without requiring any network infrastructure to carry out the verification. This scheme was published as an Internet Draft entitled 'The Named Information (ni) URI Scheme: Core Syntax' in October 2011 [FARRELL2011].

This draft was promoted in the IETF DECADE (Decoupled Data Enroute) working group that was proposing to develop an architecture for access to data stored in in-network caches. However this working group was not very successful and was closed down in September 2012 without achieving its aims.

In parallel with the DECADE work, participants in the SAIL project decided to create an experimental protocol that would allow information objects named using the ni URI scheme to be published, searched and retrieved from in-network caches. The specification for this protocol has been published as an Internet Draft [KUTSCHER2012] and the specification of the ni naming scheme in a later version [FARRELL2012] has been accepted for publication as an RFC. The resulting NetInf protocol and two convergence layers are described in Section 3. of this document.

Development of an implementation of the NetInf protocol started at a 'codesprint' during the SAIL meeting in Lisbon during January 2012. It was decided that a number of different language implementations would be prototyped to allow potential users maximum flexibility in choosing an implementation to use and to exploit the skills of the SAIL participants to best effect.

Stephen Farrell created a C language command line tool to generate ni names for Named Data Object (NDOs) as they came to be known, and provided a PHP plug-in for an Apache web server that provided a form that allowed the user of a web browser to publish an NDO using its ni name and retrieve NDOs from the cache created by the Apache server.

I created a modified version of the 'wget' command line web access client that could retrieve NDOs using the '.well-known' form of the ni name that can be used directly with the HTTP protocol to retrieve NDOs.

I then went on build prototype NetInf command line clients (**niel**, **niget**, **nipub** and **niserach**) to implement the basic message types of NetInf over the HTTP and TCP convergence layer and a standalone (rather than Apache plugin) HTTP server that was dedicated to serving NetInf protocol requests. The server could be accessed either from the command line clients or a web form as with the initial PHP server.

⁸ It would improve performance to do the same for security blocks which currently create the locals in *consume* rather than *validare*..

The initial Python implementations made extensive use of the standard Python HTTP, URLLIB and CGI libraries to handle the low level aspects of the HTTP protocol.

Subsequently the initial prototype has been improved and refactored to improve performance and allow a number of variant mechanisms to be incorporated with the core server engine. During Autumn 2012 the code was further reworked so that the HTTP server could be configured either as a standalone server or as an Apache plug-in using the Python WSGI (web Server Gateway Interface) framework⁹ and the Apache *mod_wsgi* module¹⁰. Also provision was made for the metadata cache for NDOs cached by the servers to be held in a Redis NoSQL database¹¹ as opposed to the files that were used in the earlier versions. This was done to examine the relative performance of the two schemes. The actual content is held in the filing system as individual files mainly because of the constraints on the size of the Redis database which is cached in memory during operation. Various performance tests have been carried out using the *mod_wsgi* server.

The most recent developments have been

- the addition of forwarding mechanisms to allow requests to be sent to other servers if an NDO is not found in a particular server's cache using location hints and simple name resolution information, and
- the implementation of the DTN convergence layer with the HTTP↔DTN gateway and command line tools that can use either the HTTP or the DTN convergence layer dependent on the format of the target locator supplied.

This latest code has been incorporated into the NetInf Device and was demonstrated at the SAIL final demonstration event in Kista, Sweden in February 2013. Final tidying up of this code will lead to a further release of the latest code on the Sourceforge repository where the Nilib is available as Open Source Software.

6.2 Licensing

The Nilib Open Source Software can be downloaded from the Sourceforge NetInf repository at the [NetInf Software Repository](#). The majority of this software is subject to the Apache 2 software license¹². Some small parts are modifications of standard Python software and are therefore subject to the Python Software Foundation license. Finally, two modules are subject to the MIT licence. All of these licenses are approved by the [Open Source Initiative](#) allowing free redistribution, incorporation in products and creation of derivative works.

6.3 Documentation

The Python portion of the Nilib is fully documented with extensive comments using Doxygen style markup. The documentation can be extracted from the source code using Doxygen¹³ in association with the input filter *doxypy*¹⁴. The documentation can be viewed on line at <http://tcd.netinf.eu/doc>.

6.4 Code Volume and Installation

The Python Nilib consists of about 60 original files with more than 10000 code statements. The Python code is configured as a Python module and can be built and installed using the standard Python **setuptools**. There is also a script **install-nilib-wsgi.sh** to construct an Apache virtual host file which will create a server that runs the WSGI-bases NetInf server.

9 More information on WSGI can be found at the [WSGI web site](#)

10 More information and code download for *mod_wsgi* can be found at [modwsgi - Python WSGI adapter module for Apache](#)

11 More information on Redis can be found at the [Redis web site](#)

12 A copy of the Apache 2 License can be obtained from <http://www.apache.org/licenses/LICENSE-2.0>

13 More information on Doxygen can be found at [Doxygen - documentation from Source Code](#)

14 More information on doxypy can be obtained from [Doxypy wrb site](#)

6.5 Design Considerations

This section outlines a number of design issues that had to be taken into account when creating the NILib Python code.

6.5.1 NDO Content Size and Digest Generation Efficiency

The design of NetInf assumes that NDOs will be treated as single units rather than broken into chunks or fragments. The content of NDOs may therefore be very large files. The ni URI naming scheme involves creating and/or verifying that a cryptographic hash digest of the content. It is desirable that the tools used to send and receive NDOs are able either to verify that the ni name used to access an NDO matches with the content received during retrieval or be able to generate or verify the ni name during publication.

Given that very large size content may be involved, it is likely that NDO content can not be handled purely in memory buffers and hence will have to be written directly to disk files. It would preferable if it was not necessary to reread these files in order to compute the digest. Accordingly the code incorporates mechanisms to avoid the use of large memory buffers and, wherever possible, rereading of files.

In the case of the HTTP convergence layers:

- The PUBLISH operation transmits the content from client to server. As explained in Section 3.1.1.2, the content is encoded as a MIME *multipart/form-data* structure when the content octets are to be sent with the request. For convenience the client that performs the publication may wish to construct the ni name from the content in a 'conventionally' named file rather than having the user supply the ni name as a parameter. The standard Python modules for handling HTTP clients and form data only deal with *application/x-www-form-urlencoded* format and expect that any data to be sent is in memory buffers. I have therefore created special modules that will create a *multipart/form-encoded* body for the POST request, stream the content file directly from the source file to the HTTP socket, digest the content as it is streamed to the server and send the resulting ni name built from the digest as a trailing parameter once the file has been sent. This means that the content file only needs to be read once and the digest and network operations are effectively running in parallel. Unfortunately it is not so easy to handle the reception of the content at the server so efficiently. In general the publishing server will wish to verify the digest it is supplied with but it is not generally possible for the CGI module that interprets the incoming form to know that it should be digesting the content until the whole form has been received and the form processor has interpreted the result. Thus at present verifying the digest in the server requires that the content is buffered by the incoming CGI processing and then digested as it is transferred to its permanent file in the NetInf cache run by the server. However, at least on Linux and similar operating systems, with an appropriate choice of position for the transferred file, the naming of the file in the cache can be changed to reflect the verified digest (assuming this is how content files are managed) without needing to copy the data again.
- The GET operation transmits the content from server to client. It is assumed that the server trusts the information in its cache as it is verified at publication time. Thus on the server efficient sending of content with GET-RESPONSE messages just requires the file to be streamed directly from the cache file to the network connection. In the client, a response that contains the content octets will be a three section MIME *multipart/mixed* object. The client is aware of this and can plan for the third of the three MIME objects to be a file which it has to digest. The digest algorithm to use is known from the ni name that is being retrieved so the digest can be calculated as the MIME data is read from the network. A specialized version of the Python standard email message parser has been created to read the content directly into a file and create the digest while transferring the content. As a result, the digest generated from the incoming content can be checked against the ni name as soon as the content has been received without needing a large memory buffer or rereading the data.
- The SEARCH operation is less critical since neither request nor response is expected to be very large and neither has to be subjected to the digesting process.

For the DTN convergence layer, the sorts of optimisation that have been achieved for the HTTP convergence layer are more difficult to achieve, and are probably less relevant because synchronous communications will not generally be taking place. Currently the PUBLISH operation client has to read and digest the content if it isn't supplied with the full ni name. It then passes the content file to the DTN BPA as a file and the BPA has to read the file again when sending to the network. It is not clear that this can be optimized because the two operations may be separated in time if a relevant link is not open when the publication occurs. Also at present, the BPA is not set up to verify the digest of bundles that it caches. It would be possible to construct the digest as the bundle is read from the network link but would several kinds of 'layer violations' to inform the **PayloadBlockProcessor** that it was expected to do this and would add significant amounts of NetInf specific code to the general purpose BPA. Similarly the GET operation client receives the content as a file. Verifying the digest of the content requires reading this file and digesting the content, unless the client is prepared to trust that the local BPA has checked the digest for it which could be done as the content is received from the network. Also the client has no control over the location or ownership of the bundle payload file so that it may be necessary to copy the file to a new file that is owned by the client's user and in a directory that the owner controls¹⁵. So at present the DTN CL does not attempt any optimisations with the result that content files may be read and written several times.

6.5.2 Multithreaded and/or Multiprocess Clients and Servers

If the tools provided by this library are to be of significant value in performance testing and any sort of non-trivial usage, they must be designed to provide parallel processing for requests in many circumstances.

In the case of the HTTP CL, clients and servers communicate over a TCP connection. Accordingly HTTP CL servers need to be able to handle multiple client connections in parallel. The code that has been written for the Python Nilib satisfies this requirement:

- There is a standalone multithreaded HTTP CL server that creates a new thread to service each connection (see Section 6.6.6.3)¹⁶
- The Apache 2 WSGI plug-in that is also available can operate both with multiple processes servicing requests and multiple threads within each of the processes.
- Note that the WSGI test harness does not provide either multiprocessing operation or multithreading.

The DTN CL gateway is rather less sophisticated at the BPA connection level and serialises requests and responses over (separate) single connections. It is not clear that anything much better could be achieved without modifying the way in which the DTN2 BPA API server works as there is no equivalent at the DTN EID level of the way that the TCP listener mechanism. However, once inside the gateway, any forwarded requests are handled in parallel and delays on a particular forwarded request do not impact on other requests¹⁷. Effectively the gateway also implements parallel processing of clients if a request is forwarded to several locators.

The basic command line clients only expect to contact a single locator to handle a NetInf request and so no attempt has been made to provide multiprocessing or multithreading. However test clients (see Sections 6.6.4.3 and 6.6.4.6) that are intended to issue a multiplicity of requests are written to use multiple processes, creating a new process per request using the Python **multiprocessing** module.

For the present the level of multiple request parallel processing appears to meet the requirements.

¹⁵ This is a deficiency of the DTN2 BPA which it would be useful to correct but could be difficult given that the BPA is potentially accessed by multiple different client owners.

¹⁶ There are limitations to the extent to which a multithreaded server written in Python can exploit a multi-core or multi-processor machine due to the way the Python interpreter is written. However it would be fairly trivial to convert the multithreaded standalone server into a multiprocessing standalone server by using the **ForkingMixIn** in place of the **ThreadingMixIn** (just matter of changing the base class of **NIHTTPServer**). However, this would also necessitate using the **cache_multi.py** module for the NDO cache manager as inter-process file locking is needed.

¹⁷ Unless, that is, several requests hang up at the same time and clog the available processes.

6.6 Overview of Python Nilib Modules

This section contains brief notes about each of the modules written for the Python Nilib Repository. Unless explicitly specified the modules can be found in the **python/nilib** sub-directory of the repository. The repository contains a **README** file which provides a change log and summary of what is available and the **python/setup.py** file provides the installation driver for the Python **setuptools** that will assemble the module and place it in the installation locations defined.

This document does not contain very many operational details or descriptions of the code: there are extensive descriptions of the functionality in header comments in each file and these are extracted by the Doxygen documentation tool as explained in Section 6.3.

Many of the modules contain test code in the standard Python manner whereby the test code can be exercised by running the module as a standalone program. Note that several of these modules use threading and/or network access which means that testing them within the IDLE Python development environment does not work very well.

6.6.1 Installation and Installed Scripts

The Python Nilib is configured as a single Python package that can be installed using the Python **setuptools** utility. The code has only been tested on Linux. In principle there should be no problems running on Mac OSx and at least the HTTP related aspects could be expected to run on Windows machines. However since the DTN2 BPA does not currently run on Windows, the DTN aspects cannot be used on Windows machines. Installation in the Python distribution area will probably require *root* or administrator access.

A number of the modules described in this section are intended to be run as command line utilities. Python **setuptools** provides a mechanism that automatically generates a wrapper for such modules, known as a 'console script' and installs it in one of the usual places from which user programs are executed (e.g., */usr/bin*). The following console scripts are currently installed:

- **pynicl** - see Section 6.6.3.2
- **pyniget** - see Section 6.6.4.1
- **pynigetalt** - see Section 6.6.4.2
- **pynigetlist** - see Section 6.6.4.3
- **pynipub** - see Section 6.6.4.4
- **pynipubalt** - see Section 6.6.4.5
- **pynipubdir** - see Section 6.6.4.6
- **pynisearch** - see Section 6.6.4.7
- **pyniserver** - see Section 6.6.6.3
- **pystopnserver** - see Section 6.6.6.4
- **pyniwgsiserver** - see Section 6.6.6.5
- **pyredisflush** - see Section 6.6.5.7

All these scripts use a number of command line options. They all provide a **-h/--help** option to print a usage description and definition of the available options.

HELPFUL HINT: ni scheme URIs contain the semi-colon character (**;**); this character is a statement terminator/separator in all the common command shells used by Linux or other species of Unix[®] and so it is necessary to enclose ni scheme URIs in single quotes to avoid the semi-colon being inappropriately interpreted by a command shell when an ni name is supplied as a command line parameter to any of these console scripts.

6.6.2 Logging

All modules contain extensive logging statements. In the server code this is all handled using the Python standard **logging** module but in some of the simpler command line clients, debug statements are handled via a single **debug** function that prints messages and can be disabled if not required in a production environment. See Sections 6.6.6.3, 6.6.7.3 and 6.6.6.6 for more details.

6.6.3 Core ni URI Support

Central to the NetInf protocol is the use of the ni URI naming scheme [FARRELL2012]. This section contains the core library that provides classes and functions that implement the ni naming scheme and a command line utility that will generate or verify the ni name for a piece of content contained in a file.

6.6.3.1 ni.py

This is the core library that is needed by all the other modules that handle ni names. It contains the following items:

- **Niname** class: An instance of this class encapsulates a single Ni: scheme URI. An instance can be constructed either with a single string in the correct form or a tuple of components. It knows the currently implemented digest algorithms, the truncations that are used with them, and the lengths of the resulting digests, both before and after truncation. Methods are implemented to allow checking of the syntax of a ni: scheme URI, both as a template without the digest and in complete form. The URI components can be accessed, and in most cases manipulated in order to (for example) convert it to a 'canonical form' without netloc or query string. It has a number of class methods that can be used to access fixed 'constants' within the class including the algorithm list.
- **NI** class: A 'stateless' class with a number of methods that are primarily intended for creating and checking the digests associated with a file or buffer that are incorporated into ni: scheme URIs referring to an object. A single globally accessible instance is created and made available as *Nlproc*.
- **Nidigester**: A helper class intended for use in conjunction with **encode::MultipartParam** (see Section 6.6.10.2). The point of the complexity here is to avoid either reading all of a file to be both digested and sent over an HTTP connection from a client twice or reading it into a buffer before calculating the digest needed for an ni scheme URI. When used in conjunction with the tricks in **encode::MultipartParam** and **encode::multipart_encode**, the digest can be calculated as the file is streamed out to the HTTP connection and the result incorporated into a later form parameter. The actual streaming requires the **streaminghttp** module (see Section 6.6.10.1).
- The module also exports an enumeration of errors (**ni_errs**) that can occur when parsing an ni name and a dictionary mapping these error codes to error text strings (**ni_errs_text**).

6.6.3.2 ni.cl.py (run by script pynicl)

Provides a command line utility script to either generate a ni name for a content file using a template for the scheme and digest algorithm to use or verify a ni name that contains a digest that purports to be for a content file. The utility supports the ni and nih URI schemes, the binary form of name and the .well-known HTTP URL scheme.

6.6.4 NetInf Command Line Client Utilities

The modules in this section can all be run as standalone command line programs as explained in Section 6.6.1. Each one is designed as a NetInf client to send one or more NetInf requests over the HTTP (and in some cases alternatively the DTN) convergence layer and wait for the corresponding response(s) which can then be displayed to the user or stored in a file as appropriate.

6.6.4.1 **niget.py** (run by script **pyniget**)

Basic utility to send a single NetInf **GET** request over the HTTP CL. Needs a complete ni or nih name with digest and locator (URL netloc) to which the request should be sent (i.e., a fully qualified domain name or IP address and optional TCP port number). Note that the current version does not use the performance optimizations introduced in **nigetalt.py** (see Section 6.6.4.2) that avoid reading the whole content into a buffer and generating the digest from the buffer. This utility is therefore not appropriate for very large content files. If successful, the result may be either just the metadata for the NDO or the metadata and the content octets. If the content is returned it is placed in a file. The user can select a name for the file or the program will use the ni name.

6.6.4.2 **nigetalt.py** (run by script **pynigetalt**)

Improved version of **niget.py**. Handles a single **GET** request but can use either the HTTP CL or the DTN CL if it is available. Selection of the CL to use is dependent on the scheme name in the locator specified (either **dtm://** or **http://** - defaults to HTTP if no scheme name given). For the HTTP scheme uses optimised version of the Python MIME message parser **nifedparser.py** (see Section 6.6.11.2) to allow calculation of the content digest while reading the HTTP response from the network and directing the content straight to a file rather than initially reading it into a memory buffer.

6.6.4.3 **nigetlist.py** (run by script **pynigetlist**)

Performance testing utility to issue a series of NetInf **GET** requests specified in a file over the HTTP CL only. Uses the same performance optimizations as **nigetalt.py** (see Section 6.6.4.2). Command line options allow the utility to execute each request in a separate process and collate the responses.

6.6.4.4 **nipub.py** (run by script **pynipub**)

Basic utility to run a single NetInf **PUBLISH** request. Capable of publishing either just metadata or metadata and content for an NDO over the HTTP CL. The ni name for the NDO can either be specified as a template (scheme name and digest algorithm) when the program will calculate the digest as it is streamed to the server or as a complete ni name with the scheme, algorithm and digest. Uses modules **encode.py** (see Section 6.6.10.2) to create the specialized form parameter encoding that handles on-the-fly digesting of the content and feeding the content file in chunks to **streaminghttp.py** (see Section 6.6.10.1) followed by on-the-fly generation of the form *URI* field from the digest of the content file for sending to the server. The user has to specify a locator to which to send the request (see Section 6.6.4.1 for details). The information returned with the publish response can be requested in alternative formats including a JSON string or a human readable report encoded in HTML.

6.6.4.5 **nipubalt.py** (run by script **pynipubalt**)

Extended version of **nipub.py** (see Section 6.6.4.4) that can send a single **PUBLISH** request over either the HTTP CL or the DTN CL, if it is available. Use the same optimizations as **nipub.py** (see Section 6.6.4.4). See Section 6.6.4.2 for selection of CL to use.

6.6.4.6 **nipubdir.py** (run by script **pynipubdir**)

Performance testing utility to execute a series of NetInf **PUBLISH** requests operating on the files in a directory tree using the HTTP CL only. Walks the complete directory tree and publishes either all files in the tree and as many as are specified in a count on the command line. Uses the same performance optimizations as **nipub.py** (see Section 6.6.4.2). Command line options allow the utility to execute each request in a separate process and collate the responses.

6.6.4.7 **nisearch.py** (run by script **pyniserach**)

Basic utility to execute a NetInf **SEARCH** request. The user supplies a set of tokens for the search operation to use and a locator specifying the server where the search should be sent. The user may also specify the

format of the response (either a JSON string or human readable report encoded in HTML). The choice of search mechanism and/or engine is left up to the server that processes the request. Uses the HTTP CL only.

6.6.5 Server Common Code

The modules in this section provide the common core code for all of the various HTTP CL servers that can be created, and allow the NetInf NDO cache metadata to be stored either as files or in a Redis database by means of configuration and module selection at run time.

The core of all the HTTP CL server is **nihandler.py** (Section 6.6.5.8) which has all the NetInf specific HTTP processing code for server operations. The NetInf cache management provides either for storing the NDO metadata in individual files in the filing system or in a Redis database (Section 6.6.5.6). Two variants of the file-based metadata cache cater for single process (Section 6.6.5.4) and multi-process (Section 6.6.5.5) servers. The cache mechanism used is selected at run time according to the type of server being used and configuration.

6.6.5.1 metadata.py

Provides the class **NetInfMetaData** that is used in servers to encapsulate the metadata for a single NDO. Can be constructed either from components received from clients or from NetInf **GET-RESPONSE** metadata components. There is also a method to merge additional metadata received if more than one **GET-RESPONSE** is received. The data is held as a JSON-structured dictionary and can be loaded from or dumped to one of the NetInf cache mechanisms described in Sections 6.6.5.4 to 6.6.5.6 as a JSON encoded string.

6.6.5.2 file_store.py

This is a dummy module that is imported by the server startup code to indicate to the actual server that it should be using the file-based metadata storage mechanism. The fact of this module being loaded is used to control the loading of the correct cache management module. This makes it possible in the WSGI Apache plug-in to instantiate a single instance of the cache manager on first usage in each of the child processes created by *mod_wsgi*.

6.6.5.3 redis_store.py

This is a dummy module that is imported by the server startup code to indicate to the actual server that it should be using the Redis-based metadata storage mechanism. The fact of this module being loaded is used to control the loading of the correct cache management module. This makes it possible in the WSGI Apache plug-in to instantiate a single instance of the cache manager on first usage in each of the child processes created by *mod_wsgi*.

6.6.5.4 cache_single.py

Implements the class **SingleNetInfCache** that is imported into HTTP servers as **NetInfCache** when the server is using the file metadata storage mechanism and runs as a single process but may be multi-threaded to handle the NetInf NDO cache. Because it is a single process, the cache module can manage a local in-memory cache to speed up access to metadata rather than having to access the file system every time. A single in-memory lock is used to serialize access to the cache since it is a single process.

This cache module is used by the standalone multi-threaded HTTP server (see Sections 6.6.6.3 and 6.6.6.1).

6.6.5.5 cache_multi.py

Implements the class **SingleNetInfCache** that is imported into HTTP servers as **NetInfCache** when the server is using the file metadata storage mechanism and runs as a single process but may be multi-threaded to handle the NetInf NDO cache. Because the server may be running multiple processes, this cache module cannot use an in-memory cache as well. Locking is also more complex. An in-memory lock is used to serialize access between multiple threads in each process and an operating system *flock* is held on the

metadata file while reading (shared lock) or writing (exclusive lock) the metadata file. It is not necessary to hold a file system lock on the content file because it can be created atomically by renaming a suitably positioned (i.e., on the same disk file system) temporary file and the creation operation (effectively writing the file) only takes place once. In fact because the ni name guarantees that the file contents are unique it doesn't matter if there is a race to create this file from two processes – whichever one happens to win will do just fine.

6.6.5.6 `cache_redis.py`

Implements the class **RedisNetInfCache** that is imported into HTTP servers as **NetInfCache** when the server is using the Redis database metadata storage mechanism to handle the NetInf NDO cache. The Redis storage mechanism may be used in multi-process, multi-threaded server operations. An in-memory lock is used to serialize access to the cache within each process and the discussion regarding the absence of need for an operating system lock in Section 6.6.5.4 also applies here. However, it is possible that the same Redis entry for the metadata may be simultaneously written by one process. Redis is a NOSQL database and does not provide transactional handling in the way that SQL databases (for example) understand it. Instead there is something called 'pipelining' combined with 'optimistic locking'. The technique is explained in the Redis manual on the page [Transactions](#).

6.6.5.7 `redisflush.py` (run by script `pyredisflush`)

This module is a convenience script for clearing all the data out of a Redis database in order to empty a server cache. It also deletes any corresponding content files using the location of the content cache which is stored in the database.

6.6.5.8 `nihandler.py`

This module is the core of the HTTP CL . It provides a single class **NIHTTPRequestHandler** with a considerable number of methods.

It provides the core HTTP request handler for a server managing a cache of NamedData Objects (NDOs) named with URIs from the ni scheme (**ni://..** or **nih://...**) allowing clients to access, publish or search these NDOs using the NetInf protocol over the HTTP CL.

The class implements

- NetInf protocol GET, PUBLISH and SEARCH requests over the HTTP convergence layer including handling metadata and forwarding of requests to other servers.
- Direct GETs of Named Data Objects via HTTP URL translations of ni: names.
- Various support functions including listing the cache, delivering a form to generate the POST functions and returning a **favicon.ico**
- Optionally, provision of Name Resolution Server (NRS) support, controlled by configuration file option.

The handler deals with a limited set of URLs:

- **HTTP GET/HEAD** on paths:
 - `/.well-known/ni[h]/<digest algorithm id>/<digest>`,
 - `/ni_cache/<digest algorithm id>;<digest>`,
 - `/ni_meta/<digest algorithm id>;<digest>`,
 - `/ni_qrcode/<digest algorithm id>;<digest>`,
 - `/getputform.html`,
 - `/nrsconfig.html`, (when running NRS server)
 - `/favicon.ico`, and
 - `/netinfproto/list`

- /netinfproto/checkcache
- **HTTP POST** on paths (basic system):
 - /netinfproto/get,
 - /netinfproto/publish,
 - /netinfproto/put, and
 - /netinfproto/search
- **HTTP POST** on paths (when running NRS server):
 - /netinfproto/nrsconf,
 - /netinfproto/nrslookup,
 - /netinfproto/nrsdelete, and
 - /netinfproto/nrsvals

The handler is designed so that it can be used from

- an Apache 2 *mod_wsgi* 'application' function,
- any other WSGI server framework (a simple example using the standard Python **wsgiref simple_server** reference implementation can be found in **niwsgiserver.py** - see Section 6.6.6.5), or
- via a standalone server based on the **HTTPServer/BaseHTTPRequestHandler** paradigm as implemented in **niserver_main.py** (Section 6.6.6.3) and **niserver.py** (Section 6.6.6.1).

The adaptation is handled by using a **HTTPRequestShim** from an appropriate shim module as a base class. The shim is selected at run time depending on which module imports the handler as follows:

When this module is loaded it examines the modules that have already been loaded to determine the context in which it is running. If the **niserver** module has been loaded or (for test purposes only) **niserver.py** is the 'main' module (i.e., the one that was loaded by the Python to provide the main entry point to the program), then it imports the **httpshim.py** module (Section 6.6.6.2) so that it can run as a standalone server. Otherwise it loads the **wsgishim.py** module (Section 6.6.6.6) so it can run as a WSGI server. Prior to loading this module either the **file_store.py** module (Section 6.6.5.2) or the **redis_store.py** module (Section 6.6.5.3) should have been loaded. In turn the shim modules examine the set of modules that have already been loaded to determine which cache manager module to load as explained in the descriptions of these dummy store control modules.

The shim class provides the constructor which configures the environment for each request and the *handle* method that distributes the incoming HTTP request to the correct processing method for the type of request (e.g., GET requests processed by *do_GET*, etc.). The shim also provides the methods *send_string* and *send_file* that are used to create the HTTP response.

6.6.5.9 nifwd.py

This module is used to forward NetInf requests that cannot be fulfilled by the local server to another server based on locators either passed in the affiliated data of the request or looked up in a NRS or local routing hint database. This module is still under development.

6.6.5.10 ni_exception.py

Defines and exports all the exceptions raised by modules in the Nilib modules.

6.6.6 Servers Handling the HTTP Convergence Layer

6.6.6.1 niserver.py

Implements the class **NIHTTPServer** that provides a threaded HTTP server. It is a derived class of the standard Python **HTTPServer** and **ThreadingMixIn** classes.

The **HTTPServer** class which is a subclass of **TCPServer** and in turn a subclass of **SocketServer** opens a socket that is bound to the specified port passed in the *addr* parameter to the constructor. The server then sets up a listener waiting for connection requests arriving at this socket.

The **ThreadingMixIn** from the **SocketServer** module overrides the *process_request* method in the **HTTP/TCP/SocketServer** so that it creates a new thread whenever a new connection is made and accepted. Each thread creates an instance of the **NIHTTPRequestHandler** class that processes the requests that come in on this connection.

This wrapper provides additional management for keeping track of what threads are in use and naming them for convenience in identifying logging messages, and helping with shutting down the server. It also holds a number of pieces of information as instance variables derived from the configuration file and command line arguments that will be needed by all request handler threads.

It also sets up a number of variables which can be accessed via the *server* attribute of the handler.

6.6.6.2 httpshim.py

This module implements the class **directHTTPRequestShim** which is derived from the base class **BaseHTTPRequestHandler**. This class is used as the base class for **NIHTTPRequestHandler** when the handler is used as part of the standalone threaded HTTP CL server. The main purpose of the class is to provide a wrapper for the *handle* method of the base class. The **NIHTTPRequestHandler** class and hence this class is instantiated by **NIHTTPServer** (Section 6.6.6.1) whenever the HTTP server receives a new HTTP connection. With **ThreadingMixIn** in use, these new instances are run in individual threads so that multiple requests can be processed in parallel.

This module will be loaded by **nihandler.py** as described in Section 6.6.5.8 if a standalone server is being instantiated and the **directHTTPRequestShim** aliased to **HTTPRequestShim**. This class then becomes the base class for **NIHTTPRequestHandler** in this context.

The constructor for the derived **NIHTTPRequestHandler** class remains that of the **BaseHTTPRequestHandler** as the constructor is not overloaded by either derived class..

This class isolates all the server relationships (mainly represented by *self.server*) in the *handle* method and provides the *send_string* and *send_file* methods for writing parts of the HTTP response body so that the **BaseHTTPRequestHandler** *wfile* attribute is not used directly by **NIHTTPRequestHandler**. This makes it possible to build an alternative shim that can link the **NIHTTPRequestHandler** to the WSGI interface using the alternative **swgiHTTPRequestShim** (see Section 6.6.6.6). This is done because there is no way to emulate the *wfile* route for sending the response in the WSGI interface.

6.6.6.3 niserver_main.py (run by script pyniserver)

This module is the startup script for the standalone HTTP server (**niserver.py** – Section 6.6.6.1) and also, if configured the HTTP↔DTN CL Gateway (see Section 6.6.8.1). Its main purpose is to read the master configuration file (see Section 6.6.7.2). The master configuration file provides numerous settings which can also be overridden by command line options. The available command line options can be viewed via the **-h/--help** command line option as with other scripts. The function of the configuration options is also described in the default configuration file.

Once the configuration has been read and verified this program creates an instance of **NIHTTPServer** (Section 6.6.6.1) and, if configured an instance of **DtnHttpGateway** (Section 6.6.8.1). The set of configuration values are passed to the instances for future use. Each of these instances creates one or more threads which are then set as daemons and started running.

The main thread opens a UDP socket on which it listens for datagrams (any datagram, the content is irrelevant) on the loopback interface. Receipt of a datagram triggers shutdown of the server. Shutdown would also be triggered by a signal, such as a keyboard interrupt, which terminates the blocking read of the socket. If a shutdown is triggered the main thread sends shutdown requests to the other threads that it started.

and waits for period to allow all the threds to terminate cleanly (this may rake a few seconds if a long request is in progress) and then exits the server.

Shutdown can be triggered by running the `pystopnserver` (see Section 6.6.6.4) script.

6.6.6.4 `nserver_stop.py` (run by script `pystopnserver`)

This script sends a UDP datagram with an arbitrary content to the control port (2114) on the standalone HTTP server (see Section 6.6.6.4).

6.6.6.5 `niwsgiserver.py` (run by script `pyniwsgiserver`)

This script is used to create a HTTP CL server using the WSGI interface and the reference implementation of the WSGI interface supplied with the Python distribution. The resulting server is single threaded but can otherwise use all the configuration options of the Apache plug-in that works with `mod_wsgi` in the Apache environment. It is intended as a test harness for the WSGI server implementation which can be run without having to reconfigure and restart Apache after making changes. By default it accepts HTTP requests on TCP port 8055 of the 'localhost'.

The WSGI interface requires that the plug-in defines a function simply called 'application'. This function is called for every request that the server receives.

The server can be configured by defining environment variables in the process from which it is invoked

The main function demonstrates how importing either the `file_store.py` (Section 6.6.5.2) or `redis_store.py` (Section 6.6.5.3) module is used to control the cache manager module that is imported when the `nihandler.py` module is imported.

6.6.6.6 `wsgishim.py`

This module implements the class `wsgiHTTPRequestShim`.

This module will loaded by `nihandler.py` as described in Section 6.6.5.8 if a WSGI server is being instantiated and the `wsgiHTTPRequestShim` aliased to `HTTPRequestShim`. This class then becomes the base class for `NIHTTPRequestHandler` in this context.

The class in this module provides the constructor for the derived class as the constructor is not overloaded.

When the handler is instantiated by the WSGI *application* function and its *handle_request* method is called, the (very large, at least when called via Apache) environment dictionary passed in from the WSGI framework is processed into the form needed by `NIHTTPRequestHandler`. In particular a dictionary of HTTP headers is created using the `HeaderDict` class. Other items such as Apache environment variables are processed to provide the configuration for the handler.

This shim has to implement more methods than `httpshim.py` (Section 6.6.6.2) as it is not derived from the `BaseHTTPHandler` class. The class is designed to act as an iterator that can be passed back to the WSGI framework via the *start_response* function. The iterator delivers segments of the HTTP response which were either strings supplied by *send_string* or segments of a file supplied by *send_file*.

6.6.7 HTTP Server Support Files

These files are auxiliary files used by the various HTTP CL servers.

6.6.7.1 `netinf_ver.py`

This contains a number of version strings indicating the protocol and Nilib versions that are implemented by the code. They are used to return the version of the HTTP server and to initialize items in NDO metadata.

6.6.7.2 data/niserver.conf

This file provides the default configuration for the standalone HTTP CLserver. It is read by **niserver_main.py** (see Section 6.6.6.3) during server startup. By default this configuration file will be installed in **/var/niserver**.

6.6.7.3 data/niserver_log.conf

This is the default **logging** module configuration used by the standalone HTTP CL server. Its location is configured in the main configuration file (Section 6.6.7.2) and it is read by **niserver_main.py** (see Section 6.6.6.3) during server startup. By default this configuration file will be installed in **/var/niserver**.

6.6.7.4 scripts/install-nilib-wsgi.sh

A shell script that creates and installs a virtual host file for a NetInf WSGI *mod_wsgi* server on a machine that has Apache 2.x and the *mod_wsgi* installed. It also configures **rsyslog** to direct the logging output for the NetInf handler to one of the *local* **rsyslog** streams. The script installs the latest version of Nilib Python in the machine with all necessary non-standard modules and creates directories under **/var/netinf/virtual host name>** to hold all the instance specific files, log files and the NDO cache that will be used when the server is running.

The directories are:

cache	Holds the NDO cache content files in the ndo_dir subdirectory with a sub-subdirectory per hash algorithm. If the metadata is being stored in files, the metadata is held in a parallel sub-directory meta_dir .
doc	Holds the Doxygen generated documentation tree for the Python Nilib code.
log	Holds the log files from the server.
wsgi-apps	Holds the WSGI 'application' scripts used by <i>mod_wsgi</i> . See Sections 6.6.7.5 to 6.6.7.8.
www	Static files to be delivered by HTTP CL server. See Sections 6.6.7.9 to 6.6.7.12.

6.6.7.5 scripts/test.wsgi

This is the 'Hello World!' test application script for *mod_wsgi* installations on Apache servers. It can be used to verify that the *mod_wsgi* WSGI plug-in is working. It (as one might expect) writes the string 'Hello Word!!' on the browser screen when invoked. The NetInf WSGI installation script installs this application so that it can be invoked using a path of **/testapp** on a configured server.

6.6.7.6 scripts/showenv.wsgi

This is another test application for *mod_wsgi* installations on Apache servers. It outputs the complete contents of the environment dictionary passed to the WSGI application on the browser screen when invoked. The NetInf WSGI installation script installs this application so that it can be invoked using a path of **/envapp** on a configured server.

6.6.7.7 scripts/netinf_file.wsgi

This is *mod_wsgi* application script used when an Apache NetInf virtual host is using the filesystem metadata cache mode. The following paths submitted to this server are then processed by the NetInf WSGI application:

- **/netinfproto**
- **/.well-known/ni**
- **/ni_cache**
- **/ni_meta**
- **/ni_qrcode**

The application could also process the various fixed files delivered by the server (see Sections 6.6.7.9 to 6.6.7.12) but it is more efficient to delegate these to the standard Apache static content delivery mechanism.

6.6.7.8 scripts/netinf_redis.wsgi

This is *mod_wsgi* application script used when an Apache NetInf virtual host is using the Redis metadata cache mode. The following paths submitted to this server are then processed by the NetInf WSGI application:

- /netinfproto
- /.well-known/ni
- /ni_cache
- /ni_meta
- /ni_qrcode

The application could also process the various fixed files delivered by the server (see Sections 6.6.7.9 to 6.6.7.12) but it is more efficient to delegate these to the standard Apache static content delivery mechanism.

6.6.7.9 data/help.html

Template help screen installed to assist users accessing the capabilities of a NetInf HTTP CL server.

6.6.7.10 data/getputform.html

Form code that allows browser users to use the NetInf capabilities of an HTTP CL server interactively from a browser window. Cloned from the equivalent code in the PHP implementation of Nilib. Served as a static file by the various HTTP CL servers.

6.6.7.11 data/nrsconfig.html

Form code that allows browser users to configure the routing hints for an internal Name resolution Service (NRS) interactively from a browser window. Cloned from the equivalent code in the PHP implementation of Nilib. Served as a static file by the various HTTP CL servers. This code is still under development in association with the forwarding code (see Sections 6.6.5.9 and 6.6.8.2).

6.6.7.12 data/favicon.ico

NetInf mini-icon file displayed in browser address bars and shortcut lists adjacent to links to NetInf HTTP CL pages. Served as a static file by the various HTTP CL servers.

6.6.8 Server and Gateway Handling the DTN Convergence Layer

The modules in this section are used by the NetInf HTTP↔DTN CL gateway. The gateway can be run as an adjunct to the standalone HTTP CL server (see Section 6.6.6.3) or, primarily for testing purposes, as a standalone unit using the test code in **nidtnhttpgateway.py**.

The gateway core code runs as three threads that deal with the various aspects of communication to the DTN2 BPA and executing forwarded HTTP CL requests. The threads are

- **DTN bundle receiver**: Deals with incoming DTN bundles with NetInf requests. Implemented by class **DtnReceive** in **nidtnproc.py** (Section 6.6.8.3)
- **DTN bundle sender**: Deals with sending DTN bundles with NetInf responses for requests received by the bundle receiver. Implemented by class **DtnSend** in **nidtnproc.py** (Section 6.6.8.3)
- **HTTP action forwarder**: Deals with actioning forwarded NetInf requests initially from DTN but intended to service forwarded HTTP CL requests from the HTTP CL server as well. Implemented by class **HTTPAction** in **nihttpaction.py** (Section 6.6.8.2).

Incoming requests from DTN (or, in future, HTTP) are placed in a request queue managed by **HTTPAction**. Requests are serviced using spawned processes. Potentially, a request may be forwarded to multiple locators. At the end of a selected time interval or sooner if all locators have replied, the results (if any) are compiled and a response sent. For DTN responses, a message is queued for servicing by the DTN bundle sender.

6.6.8.1 **nidtnhttpgateway.py**

This module implements the class **DtnHttpGateway**. The class constructor creates the three threads that handle the connections to the DTN2 BPA and other HTTP CL servers as described in Section 6.6.8. The methods *start_gateway* and *shutdown_gateway* are provided to control running and terminating these threads.

The test code can currently be used to run a one way gateway translating from DTN CL to HTTP CL and forwarding to other HTTP CL servers.

6.6.8.2 **nihttpaction.py**

Implements the class **HTTPAction** and provides a set of routines which are executed in spawned asynchronous processes to execute NetInf requests that results from forwarding requests queued in the class.

When a forwarding request request is queued using the *add_new_req* method, the ni name is examined to see if it has a netloc which is assumed to be accessed using HTTP. This is combined with any locators from the *loclist* in the affiliated data and the ni name is looked up in the next hop and forwarding hints Redis database to determine if there are any suggestions for places to forward the request. At present only locators accessible using the HTTP CL are considered but the intention is that there might be DTN-accessible locators as well. A list of places to forward the request is added to the list.

The *add_new_req* method can be called from other threads than the one that executes the *run* method in **HTTPAction**. The *run* method monitors the request queue and executes requests by dispatching the requests to the locators selected when the request was queued. The execution can either be done in series within the same process or asynchronous processes can be spawned to carry out the request. A pool of processes is maintained using the Python **multiprocessing** module. The *run* method tries to keep as many of the processes active as possible and is informed when a process request complete so there is a process free to execute a new request.

The results from multiple locators to which a single request was forwarded are compiled, subject to results being received within a specified time limit. For GET requests the content octets will be the same whichever locator has returned them (or something is badly wrong) but the metadata returned can be combined. The **metadata** module (Section 6.6.5.1) has a method for doing this. In the case of PUBLISH and SEARCH requests the results are just concatenated with an indication of where the results came from.

Responses to requests from the **DtnReceive** thread are queued for sending by the **DtnSend** thread.

6.6.8.3 **nidtnproc.py**

Implements the **DtnReceive** and **DtnSend** classes. Requires that there is an active DTN2 BPA on the node and needs the **dtnapi.py** module generated by DTN2 (see Section 6.6.9.1) and the other modules described in Section 6.6.9.

DtnReceive expects to receive only NetInf requests (of any sort) and possibly status reports resulting from responses to these requests. Receiving responses will be handled by separate BPA application registrations. Incoming requests are translated into an internal form as an instance of the **HTTPRequest** class (see Section 6.6.8.4) and queued for processing by calling the *add_new_req* method of **HTTPAction** (see Section 6.6.8.2).

DtnSend takes messages from the queue fed by responses in the **HTTPAction** thread. The messages are instances of class **MsgDtnEvt** and contain the updated **HTTPRequest** that was previously received and forwarded. The **HTTPRequest** is translated back into a DTN CL bundles and sent back to the source of the request.

6.6.8.4 **nidtnevtmsg.py**

Implements the **HTTPRequest** and **MsgDtnEvt** classes.

HTTPRequest is an internal form for request messages and provides fields for registering the response(s) when the message is forwarded.

MsgDtnEvt is used as the message form for events that are sent to the **DtnSend** thread for despatching as responses using the DTN CL.

6.6.9 **DTN Support and DTN2 BPA Interface**

These components assist with the interface to the DTN2 BPA. They should probably all be part of the DTN2 project. However initially only the main **dtnapi.py** module is supplied by DTN2.

6.6.9.1 **dtnapi.py**

DTN2 Python scripting interface module. This module is part of the DTN2 package rather than Nilib.

6.6.9.2 **dtn_api_const.py**

Various constants needed by the DTN2 interface but not wrapped by SWIG so not in **dtnapi.py**.

6.6.9.3 **nidtnbpq.py**

Encapsulation of a BP BPQ block. Provides translation to and from the binary on-the-wire format to an internal structure that facilitates access to the various fields.

6.6.9.4 **nidtnmetadata.py**

Encapsulation of a BP JSON Metadata block. Provides translation to and from the binary on-the-wire format to an internal structure that facilitates access to the various fields.

6.6.9.5 **nistruct.py**

Wrapper for modified Python **struct** module that can handle BP SDNV fields. Used in encoding and decoding BPQ and Metadata blocks.

6.6.9.6 **_nistruct.c**

Implementation in the C language of modified Python **struct** module that can handle BP SDNV fields.

6.6.10 **Modified 'Poster' Software**

The two module is in this section are modified versions of the 'poster' software written by Chris Atlee. The original code and further documentation is available at

<http://atlee.ca/software/poster/index.html>

This software is licensed under the MIT license which is somewhat different from the Apache 2 license but is also acceptable for Open Source releases.

6.6.10.1 **streaminghttp.py**

For this module the changes are in the documentation rather than the code.

The module is specifically designed to assist with sending *multipart/form-encoded* HTTP POST requests.

This module extends the standard `httplib` and `urllib2` objects so that iterable objects can be used in the body of HTTP requests.

In most cases all one should have to do is call `:func:`register_openers()`` to register the new streaming http handlers which will take priority over the default handlers, and then you can use iterable objects in the body of HTTP requests.

6.6.10.2 **encode.py**

This module provides functions that facilitate encoding name/value pairs as multipart/form-data suitable for a HTTP POST or PUT request.

multipart/form-data is the standard way to upload files over HTTP

This version has been modified in two ways from version 0.8.1 on the web site:

- to allow a digest for a file to be uploaded to be generated 'on the fly' as it is sent out to the network for transmission over HTTP.
- to provide values for form parameters to be generated at the time the parameter is written to the network rather than when the form is constructed. This allows values to be dependent on things (such as digests) that are calculated as the form is uploaded.

6.6.11 **Modified Standard Python Modules**

The modules in this section are modifications of existing standard Python modules taken from the Python 2.6 release. These modules are licensed under the Python Software Foundation license.

6.6.11.1 **ni_urlparse.py**

Standard Python **urlparse.py** extended with support for *ni* and *nih* URI schemes.

The module parses (absolute and relative) URLs.

Cloned from the Python 2.6 distribution `urlparse.py` on 20120130.

The module has been updated

- to add *ni*: scheme: This only required adding 'ni' scheme name to the various tables that represent the attributes of this scheme:
`uses_relative`, `uses_netloc`, `uses_params`, `uses_query`, and `uses_fragment`.
- to add *nih*: scheme: Requires adding 'nih' scheme name to the tables:
`uses_relative`, `non_hierarchical`, and `uses_params`.
We also add 'nih' to `uses_query` and `uses_fragment` although these are not allowed in *nih*. Verification weeds them out later. Also had to actually use `non_hierarchical` (which wasn't done before) to suppress the `///` in front of the path when rebuilding (in `urlunsplit`).

6.6.11.2 **nifeedparser.py**

Standard Python module **feedparser.py** in the **email** package extended to allow MIME object content to be sent directly to a file rather than being held in a memory buffer while completing the processing.

The feed parser implements an interface for incrementally parsing an email message, line by line. This has advantages for certain applications, such as those reading email messages off a socket.

FeedParser.feed is the primary interface for pushing new data into the parser. It returns when there's nothing more it can do with the available data. When you have no more data to push into the parser, call **close**. This completes the parsing and returns the root message object.

The other advantage of this parser is that it will never throw a parsing exception. Instead, when it finds something unexpected, it adds a 'defect' to the current message. Defects are just instances that live on the message object's `.defects` attribute.

For NIlb the module has been modified to allow dumping of a message direct to a file rather than into a memory buffer (good for very large files and allows generation of a digest while parsing an incoming message). An optional `_filer` parameter is given when creating the parser. This is a callable that is passed the content-type of the payload and makes a decision as to whether the message should be written to a real disk file or a **StringIO**. The result is passed to the `set_payload_dest` method of the **FiledMessage** class which is a wrapper round a standard message that manages writing the payload to a file.

6.6.12 NIlb Setup Support and Documentation

Files used by the **setuptools** mechanism and the **README** file that accompanies the distribution.

6.6.12.1 setup.py

This script is located in the python subdirectory of the NIlb distribution. It is the control script for the **setuptools** installation mechanism for the Python module.

The installation

- builds the Python package in the selected **dist-packages** directory of the Python installation in use,
- installs various package data items including configuration files, WSGI application scripts and static files served by the HTTP CL servers in the directory specified by the environment variable **NILIB_DATA_DIR** (defaults to **/var/niserver** if the environment variable is not defined),
- creates a template directory tree named **wsgi** in the **NILIB_DATA_DIR** that is copied when creating the directory tree for a Apache *mod_wsgi* plug-in virtual host (see Section 6.6.7.4).,
- creates the set of console scripts described in Section 6.6.1, and
- downloads and installs any external packages needed by the NIlb code.

6.6.12.2 __init__.py

This file is required by the **setuptools** system and can be used to initialize the module as a whole. Currently it does not contain any code but lists the modules making up the package.

6.6.12.3 MANIFEST.in

List of files to include in a distribution used by **setuptools**.

6.6.12.4 scripts/pynilib_test.sh

Shell script for testing **ni.cl.py** and **ni.py**.

6.6.12.5 README and doc/README

'nuff said!

7. Development of NetInfFS FUSE-based File System

The intention of the development of the NetInf Device described in Section 4. was to examine whether a device could be made usable in a situation where its only access to external data was through ICN using the NetInf protocol. To make this more realistic (as described in Section 1.) the NetInf device was expected to operate in a DTN environment using NetInf over the DTN convergence layer using the BP which was described in Section 3.1.2 and is implemented in the Python component of NIlb (see Sections 6.6.4.2, 6.6.4.5 and 6.6.8).

In order for existing applications to access the content of NDOs when they typically expect to have their data stored in files, it is necessary to make it easy for applications to access the content of NDOs cached on the NetInf Device as files identified by their ni name.

Similarly, applications that create data should have the opportunity for this data to be published as an NDO without needing to run extra applications to organize the publishing. They should also be able to determine the ni name that is given to the content without needing to do extra work.

7.1 Introduction to FUSE Filing Systems

The method that was adopted to allow NDOs to be read via the Linux filing system and to publish newly created files automatically was to build a pseudo-filing system using the FUSE framework.

FUSE ([Filesystem in Userspace](#)) provides a framework that allows a Linux filing system to be implemented using a userspace program rather than needing to add functionality to the Linux operating system kernel. Clearly this is much easier to develop and simpler to deploy for prospective users. The necessary kernel code is incorporated in Linux kernels from the 2.4 and 2.6 series and later.

Implementing a file system is simple, a [hello world](#) file system is less than a 100 lines long. Figure 10, taken from the FUSE web site, shows the path of a file system call (e.g. stat) in the the *hello world* example:

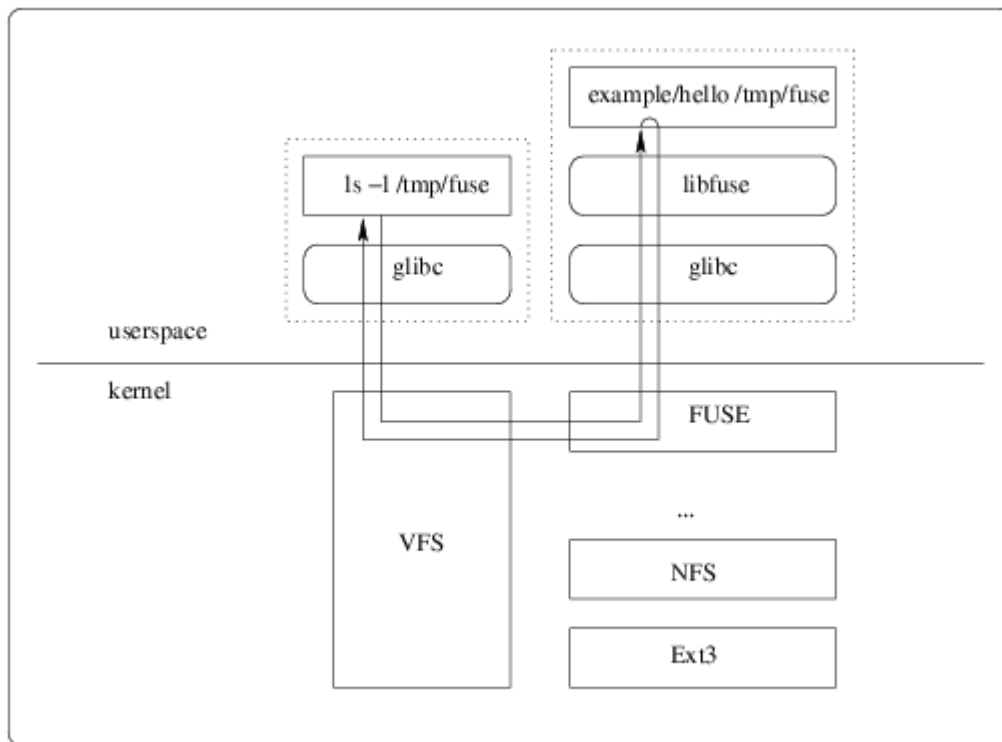


Figure 10: Control Flow in a FUSE-based Filing System

There are also wrappers that allow the userspace program to be written in various scripting languages. Since the component for the NetInf Device was intended to operate with various other component written In Python, we chose to implement the NetInfFS in Python.

As with many of the file systems that use FUSE, the NetInfFS is used to give a different view onto files in some other filing system that either already exist or to create files via a different name and/or with some additional operations. The part of the NetInfFS used to view NDO contents (i.e., DTN2 bundle payloads) can be seen as a means of automatically creating transient soft links to the bundle payload files. The 'publish' part adds the additional automatic publication operation to file creation and update. Additionally the system automatically creates an extended attribute for the file which is updated whenever the file is updated.

7.1.1 FUSE Installation Notes

The wrapper for FUSE that was used is Fuse-Python version 0.2.1. This should be downloaded from Sourceforge ([Fuse-Python Version 0.2.1 source archive](#)) rather than the version in the PyPi repository ([PyPi Fuse-Python package page](#)) due to some fixes in the xattr coding. The Canonical repository for Ubuntu 12.04 has Version 0.2.1 but for Ubuntu 10.04 has Version 0.2.

Most recent Linux distributions provide the FUSE filing system component pre-built in the kernel and have the FUSE library (*libfuse.so*) pre-installed; this is certainly the case for Ubuntu 10.04 and 12.04. If FUSE is not built into the kernel being used it will be necessary to reconfigure, rebuild and reinstall the kernel.

The NetInfFS uses extended attributes from the *user* namespace. This means that the underlying file system that is used to hold files viewed via the NetInfFS needs to be configured to allow *user* namespace attributes. This is not the default situation. Add the option **user_xattr** to the **mount** options for the appropriate file system in the */etc/fstab*. It is not necessary to reboot the system to add **user_xattr** to the options. The **remount** option for **mount**. The Linux manual page for **mount** shows how to use this option. Briefly, **user_xattr** could be merged with existing options for, say */home* by issuing the command

```
mount -o remount,user_xattr /home
```

Note that to merge the options, only specify the directory and not both directory and device.

NetInfFS also requires the **pyxattr** module to manipulate extended attributes. Version 0.5.2 or higher is recommended. It can be downloaded from [Python pyxattr repository](#) or from PyPi. This version can also be installed from the Canonical Ubuntu 12.04 repositories but the Ubuntu 10.04 repository provides Version 0.4.0 which is **not** compatible.

7.2 Implementation of NetInfFS using Fuse-Python

The bundle payload files are made accessible through a FUSE based filing system. The files can either be accessed as **<mountpoint>/bundles/<bundle ID number>** for all bundles or, if they have a BPQ block, the file is also accessible as a file in directory **<mountpoint>/ni/** with a name that matches the value of the *bpq_query* field in the BPQ block. In the usage with the NetInf Device this field will usually contain a URI with the format of the ni: URI scheme. This includes a digest of the payload file contents if it is present and non-zero in length and represents the whole bundle.

The bundle information is obtained from the MySQL database tables written by DTN2. The information we need is in the **bundles_aux** and **bundles_del** tables. These provide an optimization to allow the NetInfFS to find out which bundles have been added to and deleted from the list. The **bundles_aux** file is indexed by the *bundle_id* that is assigned to the bundle by DTN2 when the internal structure is created. The *bundle_id* is unique to the bundle and monotonically increasing. The information we are accessing is written exactly once for each bundle, so it is just necessary for the NetInfFS to keep track of the highest *bundle_id* that it has already read in and update its tables by getting any rows in **bundles_aux** with higher *bundle_id*'s than it has previously seen. Similarly the **bundles_del** table records a record for each bundle deleted with a *'create_index'* (maybe a misnomer!) which auto-increments as more bundles are deleted and entered into this table. Again NetInfFS need only worry about records in **bundles_del** with higher *create_indices* than it has already dealt with. The SQL tables are accessed through an ODBC interface using **pyodbc**. The same DSN (Data Source Name) as is used by DTN2 can be used - although it would be possible to apply additional controls to the database access by using an alternative DSN with a different user that had more restricted access to the database tables.

Experiment has revealed that there are some oddities with the **pyodbc** Python DB-API connection/cursor model. It seems to be extremely difficult to get rid of a set of rows accessed through a SELECT statement using *cursor.execute*. If the cursor is left open and another SELECT statement executed, the chances seem to be that you will either get the same set of rows (even if the database has changed) or an error response of some kind. Also connections eventually time out (MySQL has some sort of connection idle timeout). Further experiment and checking documentation indicates that starting a new connection for each update request is

not as inefficient as it sounds. **pyodbc** maintains a connection pool and it appears that a 'connection' is an internal link to a real network connection which may be reused. Further even with SELECT operations, it seems to be essential to call the *commit* function for the connection to clean out the cursor and then close cursor and connection.

Additional information about the bundles from the **bundles_aux** table is made available through extended attributes for each file (**xattr**).

The NetInfFS was initially read-only so that only a small subset of the access functions need to be implemented. The remainder were left to give their default behaviour (typically returning ENOSYS - operation not implemented). The **bundles** and **ni** directories remain read only.

A further addition in version 1.4 has been to provide a directory 'publish' into which files can be written and when the file is closed after writing it is automatically published by submitting a NetInf **PUBLISH** request to the DTN2 BPA for which bundles are being displayed. In this process the ni name for the file is created and the name made available via an extended attribute called '*user.ni_url*'.

The directory 'publish' is a view onto a conventional filing system directory used to hold the published files. If the file is updated, a new digest and corresponding ni name will have to be calculated and the file republished.

The value associated with '*user.ni_url*' is a JSON encoded string with an object that contains a single entry named 'published' with an array of the *ni_url*'s under which the file has been published.

The metadata supplied to the NetInf publish request ties the object to the actual file name.

The implementation of NetInfFS is a single Python module **netinffs.py** based on the example **xmp.py** supplied with Fuse-Python. It uses a function from the **nipubalt.py** client (see Section 6.6.4.5) to perform the automatic publication of

7.2.1 Starting and Stopping NetInfFS

As noted previously, the NetInfFS is a single module of code which can be run as a Python script.

It requires that the Nilib Python code is installed and that the MySQL database used by DTN2 has been initialized. However it is not essential that the DTN2 BPA is running when the NetInfFS is started.

Before starting **netinffs.py**:

- Ensure the MySQL database used by **dtnd** (the DTN2 BPA) is initialized.
- Know the DSN (Data Source Name) used by ODBC with **dtnd**
- Create or know of an (empty) writeable directory to use as the NetInfFS mount point
- Create or know of a directory which is writeable and searchable by the user(s) who will be using the NetInfFS for the *publish* directory.
- Know the directory in which **dtnd** stores its bundle files.

The command line options used by **netinffs.py** can be checked by using the -h/--help option.

By default, **netinffs.py** 'daemonizes' itself on startup. To assist with debugging, it can be run in the foreground by using the hidden -f option. There is also a --debug option which provides additional logging output.

For debugging purposes, a log file can be written in the directory where the program is started.

Once running, **netinffs.py** should run indefinitely.

It is stopped by unmounting the filing system using the command

```
fusermount -u <mountpoint>
```

7.2.2 Handling Bundle Fragments

*** This functionality is not yet implemented and we currently only deal with complete bundles.

Note that a bundle may be in fragments. Consideration needs to be given to how to 'read' such a bundle file. When accessing it through the bundles directory, we should treat it as a sparse file (i.e., one that contains areas that will read as all zeroes - 'holes' – [Wikipedia article on sparse files](#)). For a single bundle this looks straightforward - we know the offset and size of the fragment as well as the total size of the bundle.

For bundles accessed via their *ni* name, the situation is more complex. Multiple bundles will carry fragments with the same *bpq_query* name but will contain assorted chunks of the payload file. The internal data structure in the NetInfFS need to manage a list of bundles associated with the name and handle reading the chunks by accessing the appropriate file. Thought has to be given to optimising the opening and closing of multiple files if the reading wanders back and forth through the file. It is probably best to cache some file descriptors, but we can just have one open at a time to see how it works. The file size of the file should be reported as the payload size rather than the size of a given bundle. Note that checking that the 'fragments' are part of a single bundle needs the same algorithms that are used in building the BPQ cache in DTN2 - because response fragments may come from different sources and are generated at different times the basic identifier of the bundle doesn't help. We need to look at the bundle identifier in the BPQ block (and this has to be in the data put in the **bundles_aux** table).

7.2.3 Future Optimisation

Experimentation has revealed that FUSE often makes a several calls into the Fuse-Python program to execute a single system call. The initial version of **netinffs.py** is configured to be single threaded and checks the database for updates on most calls into the Fuse-Python program to check attributes and directory contents. This works but is slow and inefficient.

The system can be improved by running a timer thread that checks the database periodically (say, every second) but not on every call into the program. This would require implementing a lock on **netinffs.py**'s internal cache of information to avoid corruption when accessed by multiple threads but would make the program more friendly to **netinffs** filing systems as it could be run in multithreaded mode.

8. Analysis and Conclusions

Developing the formal specification and implementation of the NetInf protocol followed by deployment in a device that uses NetInf as its major communication mechanism has achieved a number of goals

- Demonstrating a practical implementation of an ICN architecture protocol that does not rely on a universally deployed PKI (Public Key Infrastructure)
- Demonstrating the interdomain applicability of the NetInf Protocol.
- Demonstrating the NetInf Device using the NetInf Protocol as its sole communication means and identifying the limitations of this approach with current applications and API.
- Providing a reasonably robust implementation of the NetInf Protocol with HTTP and DTN BP convergence layers and making this available as Open Source Software.
- Improving the DTN2 reference implementation of the BPA and BP to fully support scripting languages and operate correctly with extension and metadata blocks.

8.1 Demonstrating the NetInf Architecture

Building a large scale ICN infrastructure requires a number of engineering trade-offs that have to be balanced with the requirements that we wish to place upon the system. The fundamental requirements appear to be:

- A user retrieving some content by name wants to be sure that the content received matches the name by which it was found.
- The user does not need to care from where the content was retrieved.

To satisfy these requirements and still provide a useful system on a large, possibly global, scale it is essential that

- publication of content, including generation of the names for the content, is simple,
- retrieval of the content is straightforward and efficient,
- the system does not introduce bottlenecks or single points of failure into the network, and preferably removes some existing ones,
- the system should work across various network technology domain boundaries, and
- the system is incrementally deployable so that it can be added to the existing network and use existing technology at least as initial infrastructure.

The NetInf architecture using the ni URI naming scheme inherently satisfies the fundamental requirements in that the name incorporates a secure cryptographic hash digest of the content into the name together with the name of the hash used. The scheme also has the additional advantage of being security algorithm agile, so that should the hash algorithm (usually SHA256 in the current work) become compromised in the passage of time due to improved attack algorithms or faster processors, it can be replaced immediately.

The implementation that has been carried out indicates that both publication and retrieval are straightforward. Generation and verification of ni names can be carried out with standard, readily available and well-attested software. Performance measurements that have been carried out on the implementation indicate that the major time cost penalty as compared with using a standard HTTP server is the calculation of the digest either for publication or retrieval. The current implementation carries out these operations entirely in software, using the OpenSSL libraries¹⁸. Production deployments of a NetInf-like architecture would be able to take advantage of hardware acceleration to largely overcome this penalty using either dedicated security processors such as the [VIA Padlock Security Device](#) built into some modern processors or security algorithms implemented in graphics processors.

Importantly, both generation and validation can be carried out locally without reference to any infrastructure such as a global PKI system. Verification can be considered as a kind of zero-knowledge operation in which the name provides all the information needed to verify the content. This means that the system can be deployed incrementally and does not require additional work to make it useful.

The ni naming scheme provides a novel combination of a flat naming scheme via the digest in the name which allows any instance of the named content (the NDO) to be identified immediately – identification requires only comparison of the digest algorithm identifier and the digest value – with a flexible location scheme that can be used to perform node local searches within just the current node, a local network search using a multicast mechanism or a wider area using a name resolution system to identify an instance of the NDO sought. This scheme also assists the routing of requests through possible disparate network domains. The forwarding system and the interdomain gateway that has been implemented demonstrates the practicality of this scheme. Parallel work in other parts of the SAIL project have extended the name resolution aspects in various ways that could be expanded to a global system.

The development of interoperable convergence layers suitable for operation in the highly connected Internet domain and the likely communication challenged DTN domain shows that the NetInf architecture can be used in such a hybrid environment as envisaged in the original proposal. The CL approach appears to be a suitable solution but there are some potential downsides related to the granularity of transfer with NDOs. This is discussed further in Section 8.4 below.

¹⁸ The [OpenSSL Project web site](#)

It should also be pointed out that the zero-knowledge verification scheme clearly facilitates retrieval from any location that has a copy and minimizes the need for references to any central system or location to verify the integrity of the content, but this does have the downside that it does not inherently offer any information to the user regarding the provenance of the NDO and limits the ability of the publisher to track the usage of an NDO once it has been pushed out into the network.

8.2 The Rôle of Affiliated Data and Transmitting Alternative Forks

In the course of developing the NetInf protocol the rôle of what is usually described as metadata became proportionally more important as work progressed. It became clear that the data that was 'affiliated' to the NDO could be roughly subdivided into three classes:

Technical Data such as the content type of the NDO and the size of the data.

Historic Information about how it has been handled by NetInf such as where it is stored and when it was published in these places.

Ontological Information about the information in the NDO such as an abstract and information about searches that generated a match with this NDO.

There is some overlap between these categories and there would doubtless be discussion about where certain items should be classified (for example, is the creation date of the NDO technical or historic information?). The important thing is that not all of this information is fixed when the NDO is initially created – clearly some of it is, but some information is added as the NDO is handled by the NetInf protocol. Some of this is very useful for the effective operation of the NetInf such as locations where the information is published that can supplement information obtained from a Name Resolution System (NRS). It may also be important in communication challenged networks, such as in DTN situations, where avoiding the necessity for an extra access to another system running the NRS may save a considerable amount of time.

Similarly having ontological data available may allow search operations to be short circuited if there is a match or, at least, an overlap between the search tokens in a new search and a recorded search in the ontological data.

The NetInf protocol now uses the term 'affiliated data' to cover both the metadata that is explicitly about the content (see, for example, [RFC5013]) and other information that covers how the NDO has been handled by the protocol.

The NetInf protocol has been developed to allow the carriage of this affiliated data; the ni URI scheme allows a user to verify the content but does not provide any information about the authenticity of the affiliated data. It seems that further research is needed into ways that users can place a higher level of trust in both the metadata and, preferably, the other parts of the affiliated data in an ICN.

It is interesting to contrast the handling of information in storage systems such as computer file systems with the treatment of information in an ICN. Many file systems cater for the storage of affiliated data in association with the actual content data. The terms that are used to cover this paradigm are 'forks' and 'extended attributes'. Forks are hardly a new concept: Macintosh and the Acorn ADFS filing systems from the mid-1980s introduced the concept. The Windows NT filing system apparently has a fork capability but it is hardly 'front and centre'. It is gradually filtering into Linux filing systems in the form of extended attributes but again the capability is not used significantly in mainstream applications. In all probability, this lack of integration into applications other than in the Apple Macintosh series of computers is key to this neglect (indeed it can be a security risk if applications and systems are not properly aware of it). It seems appropriate to consider paralleling the development of forks into ICN systems so that forks other than the data fork can be transmitted.

Making the affiliated data accessible to search operations and, conversely, ensuring that searches take the affiliated data into account also appears to be a fruitful area for study in ICNs.

8.3 Multiple Results and Merging Affiliated Data

The NetInf protocol is designed to allow a request to be directed along multiple paths to more than one destination and NDOs are expected to be cached in multiple separate nodes.

In the case of **GET** requests this may elicit multiple responses containing either just affiliated data or the affiliated data and the content data. When multiple responses arrive at a node, the results can be merged. In the case of the content data, each of the responses contains the same data, assuming it hasn't been corrupted in transit or by a malicious node. If the node is paranoid, it can check the digest of the data to ensure it is forwarding a good copy. The affiliated data, on the other hand, will generally be different in the various responses, certainly in respect of the historic data, and it may also contain different ontological data if different searches have been carried out on particular cached instances. The responses will all be in the form of JSON-encoded strings. It is possible to merge the affiliated data producing a union of the historic and ontological components and retaining a single copy of the static technical data. This merged form can be cached locally and forwarded as a single result towards the request source.

In the case of **PUBLISH** and **SEARCH** requests, the responses constitute separate reports from the nodes that actioned the request (if they have anything to report). It is probably appropriate to concatenate the responses indicating the source of the report in each case.

This merging operation is a function of the NetInf layer at each NetInf-capable node. It is currently not covered by the NetInf protocol specification. There is a prototype implementation in the HTTP↔DTN gateway and in the HTTP server forwarding code. The specification needs further discussion and should then be incorporated into the protocol, specification.

8.4 Granularity of NDOs

The NetInf architecture and protocol envisage that the Named Data Objects published and retrieved through the protocol are complete logical units of data. This choice of granularity for NDOs contrasts with other approaches such as Named Data Networking (NDN)¹⁹ which name data at the datagram granularity. The NetInf protocol does not support fragmentation of NDOs during transmission.

The NetInf approach is more closely matched to the storage paradigm for information in devices rather the NDN approach which is more closely matched to the transport mechanisms in the Internet. However for use in network domains such as DTN networks, the NetInf approach matches more closely to the underlying transport model where 'bundles' are also expected to be complete units of information.

The larger granularity and mapping to complete units of information in NetInf makes the addition of affiliated data to NetInf messages a more realistic proposition. Addition of affiliated data to NDN messages would probably be a prohibitive overhead if every message had to carry the affiliated data. However, how the attachment of affiliated data to NDN messages could be achieved has not been studied so this remains conjectural.

The downsides of the selected granularity and the requirement that NDOs are treated as indivisible units at the NetInf layer²⁰ include:

- real time streaming of content is not directly supported but SAIL has developed a scheme to provide streaming under the NetInf architecture, and
- using a ni name for dynamically created content requires additional work. A scheme for extending the ni scheme to dynamic content is described in Section 2.1 of [HALLAM2012]

¹⁹ [Named Data Networking web site](#)

²⁰ Note: this does not rule out a CL transporting the NDO in smaller chunks but it has to be reassembled when it reaches the next NetInf node.

8.5 The NetInf Device

Due to a number of unforeseen problems with the development of the infrastructure not as much time as would have been desirable was able to be devoted to experimentation with the NetInf Device during this project. However the basic aim of being able to access NDOs through the file interface was achieved and it was possible to write files for automatic publication.

Fully integrating retrieval of NDOs with existing applications was not attempted. However, previous work performed by our group during the previous N4C project indicated that a web interface could be provided that would allow users to request retrieval of NDOs and notify them when they had arrived. This mechanism would allow NDOs to be retrieved and cached locally and thus made available to applications through the filing system interface. The NetInf **SEARCH** request would allow users to determine if there were NDOs that would be useful. Again the web interface implemented to display search results from NetInf would assist in informing users of relevant NDOs to retrieve.

As a result of the experimentation it was clear that a number of adaptations would have to be made if ICN was to be fully usable as the sole communication means for the device:

- Access to dynamically varying internal data of the device from outside (e.g., for management or direct access to devices) was not tested but the scheme suggested in the extensions of the ni URI scheme in Section 2.1 of [HALLAM2012] would provide a way of accessing such dynamic data using a hashed public key as the name of the data.
- The socket abstraction is not an ideal API for accessing NDOs that are not already present on the device. If an application knows the ni name for an NDO but it is not yet cached in the device, a NetInf **GET** request could be issued automatically when an attempt is made to read the data. However if the data has to be retrieved via the DTN network, it may be some time before the data becomes available (this may be true even for access over the Internet of course depending on where copies of the NDO are currently cached). A notification mechanism would be preferable so that the application could continue with other processing while the **GET** operation proceeds and be informed when the data was available. This would require significant alteration to the user interface of many applications as well as education of users to expect such delays. Experimentation with a Dbus API for NetInf might offer some useful information as to how this mechanism could be implemented.
- Currently applications do not in general take into account any information forks or extended attributes of NDOs (i.e., files). The importance of affiliated data for NetInf NDOs would make it desirable for applications to be able to display and/or use some or all of the affiliated data.
- In many applications (such as writing this document), multiple versions of the document are written either as snapshots to protect against losing work or when publishing work in progress. The automatic publishing interface created in the NetInf Device provides a way to associate multiple ni names with a single file object but this mechanism needs further refinement. The mechanism is in some ways akin to an automated attachment to a revision control system or perhaps more closely (and intriguingly) a 'versioning' filing system such as the one originally developed for the Digital Equipment Corporation VAX/VMS operating system and currently maintained as OpenVMS by Hewlett Packard²¹.

8.6 Practical Utility of the Development

In practical terms, the development of a reasonably robust implementation of a new experimental protocol in parallel with the specification of the protocol has been instructive. The feedback loop between the implementation and the specification has a salutary effect on both pieces, making for a simpler, less overloaded specification and helping to give a closer match between specification and implementation. The development of 'running code' in parallel with a specification is intended to be a guiding principle of the IETF where the specification has been published, but this has become less well followed in recent years and

²¹ [Hewlett Packard OpenVMS documentation](#)

it is helpful both from the point of view of research and as a guide to standardization to participate in this effort.

9. Suggested Further Developments

9.1 Netinffs Further Work

- Cope with long delays on **PUBLISH** responses
- Complete the history/link to bundle in current version
- Move database checking to a thread so it isn't done too frequently
- Implement auto-get so that reading non-existent ni URIs in **ni** directory automatically fires off a NetInf **GET** operation for the NDO.
- Provide NDO metadata as extended attribute for bundles with BPQ blocks (see Section 9.3).

9.2 NetInf API

- Build a DBus interface to NetInf as suggested (by me) at ICNRG meeting.
- Add explicit timeout value to requests so that interfaces can respond tidily if the request takes longer to service over DTN than web browsers and/or humans can tolerate.

9.3 DTN Specification

- Finish BPQ spec
- Work out whether **PUBLISH** belongs in BPQ block
- Do specification (I-D) for JSON Metadata block
- Think about whether payload placeholder Metablock is really needed.
- If bundle with BPQ block also has JSON ontology Metadata block, put the JSON string into the bundle auxiliary table data also.
- We observe that there is no way for a recipient of a bundle to know if all the extension and/or metadata blocks that were sent have arrived. If they are deliberately or accidentally deleted its impossible to tell. Its also impossible for a custodian to know if what it has in custody matches what the originator sent or whether it has lost or gained extension blocks.

9.4 NetInf DTN

- Sort out Replicate Block flags and ensure that BPQ and Metadata blocks are not dropped because a node doesn't have a BPQ or Metadata Block Processor.
- Think about block ordering in DTN bundles: There is an issue in the API in that *all* API blocks are unconditionally placed before the payload currently and in reverse order of placement in the API. This currently doesn't really matter but it would be good to have some control if extension blocks are to be really useful.. At the moment generated Metadata blocks always go on the end of the bundle. Will probably live with this since there aren't any currently? But does this interact with the BSP?
- A general problem with DTN BP and DTN2 is that there is no way of ensuring that extension and metadata blocks put into a bundle on creation actually make it to the destination and there is no way for the destination (or any intermediate node) to be sure that the bundle it receives has the same set of blocks that it had when it was created. This is potentially an issue for nodes that offer to take custody. The BSP is no help here although the BAB makes sure the bundles is not corrupted per single hop but this doesn't

help if a node deletes blocks inappropriately (or because the user forgot to set the 'pass through if not recognized' block flag.

Bibliography

- BPQ2012 Farrell, Stephen; Lynch, Aidan; Kutscher, Dirk; Lindgren, Anders, "Bundle Protocol Query Extension Block", IETF, May 2012, <http://tools.ietf.org/html/draft-irtf-dtnrg-bpq-00>
- DTN2SQL Davies, Elwyn, "Improvements and Additions to the DTN2 Reference Implementation of the Bundle Protocol (BP) and the Oasys Framework Adding SQL Database Storage and Auxiliary Tables", TCD, February 2013,
- FARRELL2011 Farrell, Stephen; Kutscher, Dirk; Dannewitz, Christian; Ohlman, Borje; Hallan-Baker, Phillip, "The Named Information (ni) URI Scheme: Core Syntax", IETF, October 2011, <http://tools.ietf.org/html/draft-farrell-decade-ni-00>
- FARRELL2012 Farrell, Stephen; Kutscher, Dirk; Dannewitz, Christian; Ohlman, Borje; Keranan, Ari; Hallan-Baker, Phillip, "Naming Things with Hashes", IETF, August 2012, <http://tools.ietf.org/html/draft-farrell-decade-ni-10>
- HALLAM2012 Hallam-Baker, Phillip; Stradling, Rob; Farrell, Stephen; Kutscher, Dirk; Ohlman, Börje, "The Named Information (ni) URI Scheme: Optional Features", IETF, June 2012, <http://tools.ietf.org/html/draft-hallambaker-decade-ni-params-03>
- KUTSCHER2012 Kutscher, Dirk; Farrell, Stephen; Davies, Elwyn, "The NetInf Protocol", IETF, February 2013, <http://tools.ietf.org/html/draft-kutscher-icnrg-netinf-proto-01>
- LYNCH2012 Lynch, Aidan, Using the BPQ Block for a DTN-based Search Mechanism in DTN2 , 2012
- RFC5013 Kunze, John A.; Baker, Thomas, The Dublin Core Metadata Element Set, 2007
- RFC5050 Scott, Keith; Burleigh, Scott, "Bundle Protocol Specification", IETF, November 2007, <http://tools.ietf.org/html/rfc5050>
- RFC6258 Symington, Susan, "Delay-Tolerant Networking Metadata Extension Block", IETF, May 2011, <http://tools.ietf.org/html/rfc6258>
- SCOTT2011 Scott, Keith, "Changes to the DTN2 Implementation to Support ARMY VRL Router v5", Mitre Inc, July 2011,

Revision History

Version	Date	Author	Comments
0.0	20/02/13	Elwyn Davies	Creation
0.1	28/02/13	Elwyn Davies	More creation
0.2	01/03/13	Elwyn Davies	More creation
0.3	04/03/13	Elwyn Davies	More creation
0.4	05/03/13	Elwyn Davies	More creation.
0.5	06/03/13	Elwyn Davies	More creation (Nlib part 1)
0.6	07/03/13	Elwyn Davies	More creation (Nlib part2)
0.7	08/03/13	Elwyn Davies	More creation. (NetInfFS)
0.8	08/03/13	Elwyn Davies	More creation. (analysis)
0.9	12/03/13	Elwyn Davies	More creation. (analysis)
1.0	12/03/13	Elwyn Davies	Added Executive Summary. Initial release for comments
1.1	13/03/13	Elwyn Davies	Added section on multiple results. Improved Executive Summary.
1.2	21/03/13	Elwyn Davies	Minor typos fixed
1.3	14/06/13	Elwyn Davies	Added header page for publication as SCSS technical report.