NetInf Node for Bluetooth Enabled Android Devices

Linus Sunde
Abstract

NetInf Node for Bluetooth Enabled Android Devices

Linus Sunde

Information-Centric Networking (ICN) is an alternative to today’s Host-Centric Networking, such as TCP/IP. It tries to solve content delivery problems by naming data instead of hosts. This allows for, among other things, effective in-network caching and request aggregation. Network of Information (NetInf) is an ICN architecture.

A prototype providing NetInf functionality to Android applications was developed during a previous master’s thesis and project course. The prototype consists of a separate Android application. Taking into account lessons learned during the development of the prototype, it is rewritten as a library.

The new library is designed to be easier to use and extend than the previous prototype. To achieve this, the dependency on the OpenNetInf library is removed. The new library is extended with support for request aggregation. A Bluetooth Convergence Layer is also specified and implemented to support communication over Bluetooth, but could potentially be used over any protocol providing a reliable bitstream, e.g., TCP/IP. To demonstrate the usage of the new library, an application allowing for live video streaming is implemented.

Measurements are performed in order to investigate properties of the Bluetooth Convergence layer implementation. Approximately 200 KIB/s of bandwidth is available in the ideal case. Small objects are transferred at lower rates, due to proportionally larger overhead. The setup time of a Bluetooth connection is in the order of seconds, indicating that the library’s support for connection reuse is necessary.
## Contents

1 Introduction ................................. 1
   1.1 Organization of Thesis ..................... 2

2 Background .................................... 3
   2.1 Information-Centric Networking .............. 3
   2.2 Network of Information ...................... 3
      2.2.1 Named Data Objects ..................... 4
      2.2.2 Protocol .................................. 5
      2.2.3 Convergence Layers ...................... 8
   2.3 OpenNetInf ................................ 9
   2.4 Android NetInf Prototype .................... 9

3 Android NetInf Library ....................... 11
   3.1 Design ..................................... 11
   3.2 Implementation ............................. 12
      3.2.1 Ndo .................................. 13
      3.2.2 Message ................................ 14
      3.2.3 Node .................................. 16
      3.2.4 PublishController ....................... 19
      3.2.5 GetController ....................... 19
      3.2.6 SearchController ....................... 20
      3.2.7 LogController ........................... 20
      3.2.8 Api .................................. 20
      3.2.9 PublishService/GetService/SearchService 21
      3.2.10 LogService ........................... 21
   3.3 Usage Example ............................. 22

4 Bluetooth Convergence Layer ................. 25
   4.1 Specification .............................. 25
      4.1.1 Assumptions ............................. 25
4.1.2 Messages ........................................... 26
4.2 Implementation .................................... 26
  4.2.1 Android Bluetooth API ....................... 26
  4.2.2 Overview ..................................... 27
  4.2.3 BluetoothApi .................................. 29
  4.2.4 BluetoothDiscovery ......................... 29
  4.2.5 BluetoothServer ............................. 29
  4.2.6 BluetoothSocketManager .................. 29
  4.2.7 BluetoothSocketHandler ................. 30
  4.2.8 BluetoothPublish/Get/Search ............ 30

5 Live Video Streaming .................................. 31
  5.1 Dynamic Content .............................. 31
  5.2 Live Video Streaming Application .......... 33

6 Evaluation and Conclusions .......................... 36
  6.1 Measurements .................................. 36
  6.2 Discussion .................................... 39
    6.2.1 Bluetooth Convergence Layer ............ 39
    6.2.2 Live Video Streaming Application ...... 40

7 Future Work ............................................ 41
  7.1 Android NetInf Library ..................... 41
  7.2 Bluetooth Convergence Layer ............... 42
  7.3 Live Video Streaming Application .......... 42

References ............................................. 43

A Terminology .......................................... 47

B Measurements ........................................ 49

C The NetInf Bluetooth Convergence Layer ....... 53
Chapter 1

Introduction

Information-Centric Networking (ICN) [1] is an alternative to today’s Host-Centric Networking (HCN), that is one-to-one communication such as TCP/IP. ICN focuses on retrieving data instead of from where to retrieve it. That is, a request for data can be answered by any node with the data available. This aims to provide peer-to-peer functionality and caching as part of the network, in an attempt to among other things, reduce bandwidth usage.

Flash crowd events are a problem in current mobile networks. These can, for example, occur when a spectator at a stadium is filming something and everyone else wants to watch it. Todays HCN is not designed to handle such content distribution. In the mobile device scenario, this could lead to congestion in the 3G network. As these one-to-many scenarios make up most of the Internet traffic today, more efficient ways of handling them are needed.

Network of Information (NetInf) [2–4] is one version of ICN. NetInf has been designed to be able to work on top of current network infrastructure, but not depend on it.

A prototype implementation of NetInf for Android devices was developed as part of a previous master’s thesis [5] and further extended during a project course [6]. This implementation uses parts of OpenNetInf [7], an earlier NetInf implementation in Java.

The existing prototype implements the necessary functionality to support the applications that were developed in parallel with it, but it is not as general as it could be. For example, while the prototype is able to send files over Bluetooth [8], it does not do so by sending NetInf messages over a
Convergence Layer (CL) which is needed to provide all expected functionality. Reusing the prototype for other applications also require more work than should be needed. Moreover, there is also a problem with the prototype getting killed by the Android memory management to free up memory while it is running in the background, resulting in the loss of NetInf functionality.

In an attempt to improve the existing prototype and evaluate the result:

- The implementation is rewritten from scratch as an Android library using a design similar to that of the previous prototype. The memory problem is avoided because of the library implementation.
- The OpenNetInf dependency is removed.
- Support for request aggregation is added.
- Support for the Bluetooth Convergence Layer (BCL) is added.
- Some preliminary measurements are performed to evaluate the BCL performance.
- To demonstrate the usage of the new library, the video streaming use case mentioned above is considered. A simple application supporting live streaming video using NetInf is implemented using the library.

### 1.1 Organization of Thesis

Following the introduction in Chapter 1, Chapter 2 describes ICN in general and NetInf in particular. The existing prototype and its problems are also outlined here. Chapter 3 describes the design and implementation of the new Android NetInf Library. The BCL specification and implementation are presented in Chapter 4. In Chapter 5 the live video streaming use case is discussed and a simple implementation described. Measurements performed to evaluate the implementations are presented and discussed in Chapter 6. Finally, Chapter 7 presents possible future extensions and modifications.
Chapter 2

Background

2.1 Information-Centric Networking

As mentioned above, ICN is an alternative to HCN. It is a current area of research, and aims to create a network infrastructure that fulfills today’s demands. Two important aspects are content distribution and mobility. The HCN (one-to-one) approach is not well suited for today’s Internet where content, such as video, is served to large user bases (one-to-many). To meet these demands, ICN provides caching built into the network, peer-to-peer communication, and only cares what data is retrieved (not from where it is retrieved). There are several versions of ICN in active development [9]. Four of these are: Data-Oriented Network Architecture (DONA) [10], Content-Centric Networking (CCN) [11], Publish-Subscribe Internet Routing Paradigm (PSIRP) [12], and NetInf [2–4].

2.2 Network of Information

The existing prototype uses NetInf. NetInf first saw the light of day during the FP7 4WARD Project [13, 14]. Development continued in the FP7 SAIL Project [15, 16].
2.2.1 Named Data Objects

Named Data Objects (NDOs) is a common abstraction among different ICN approaches, but the details differ. An NDO can for example be a web page, a photo, or a streaming video. That is, an NDO is a block of arbitrary information stored on a computer. The name of an NDO does not change depending on where or how it is stored. This means that two NDOs with the same name are for all intent and purposes equivalent. A request for an NDO can be satisfied by any node that has a copy of an NDO with the same name.

In NetInf, NDO names use the Named Information (NI) naming scheme [2, 17, 18]. Most commonly used is the “ni” Uniform Resource Identifier (URI) scheme. A NI URI has the following structure:

\[ \text{ni://[Authority]/<Digest Algorithm>;<Digest Value>? <Query Parameters>]} \]

[Authority] is an optional field that may be included to assist applications in finding the NDO.

<Digest Algorithm> is a mandatory field detailing the digest algorithm applied to the binary data of the NDO to produce the digest value. It is mandatory for NetInf nodes to support the SHA-256 [19] digest algorithm.

<Digest Value> is a mandatory field with the output from the digest algorithm. The digest value must be encoded using base64url [20], with any padding “=” characters removed.

The <Digest Value> may be followed by the query parameter separator “?” and a key=value list of optional query parameters. These uses the same form as in HTTP URIs [21].

Two NDOs are equivalent if their names have the same digest algorithm and digest value parts.

Consider a file containing the text “hello world” encoded using UTF-8 [22]. The binary content of the file is hashed using the SHA-256 algorithm. In this example the NDO is then published by the authority “example.com”. This would result in the following NI URI:

\[ \text{ni://example.com/sha-256;}qUiQTy8PR5uPgZdpSzAYSw0u0cHNKh7A-4XSmaGSpEc \]

Remember that two NDOs are equivalent if their names have the same digest
algorithm and digest value parts. For example the following name represents the same NDO as the above example:

\texttt{ni:///sha-256;qUiQTy8PR5uPgZdpSzAYSw0u0cHNKh7A-4XSmaGSpEc}

The fact that the name can be generated from the content provides name-data integrity. That is, given a name and some data, it can be verified that the given data belongs to the given name.

### 2.2.2 Protocol

The NetInf protocol [2, 3] is message based and consists of three basic requests: GET, PUBLISH, and SEARCH. As well as their corresponding responses: GET-RESP, PUBLISH-RESP, and SEARCH-RESP. Common for all types of requests is that they should contain a message identifier (msg-id). This msg-id should not contain any personal information and should be chosen so that collisions are very unlikely to occur. According to the protocol specification [3], a long enough random string should be used. The corresponding response should then contain a status code and the same msg-id. A high level description of how NDO requests might be handled can be seen in Figure 2.1 and 2.2.
Figure 2.1: Handling an NDO request using forwarding [6]
1. I have NDO Gq9gz...

2. Please give me NDO Gq9gz...

3. Sorry, I don’t have NDO Gq9gz...
But the guy with the cap does!

4. Please give me NDO Gq9gz...

5. Here, have NDO Gq9gz...

6. When I hash the NDO data I get Gq9gz...
  This is the same as the name, I got the correct data!

Figure 2.2: Handling an NDO request using locators [6]

GET/GET-RESP

A GET request is sent in order to retrieve an NDO from the NetInf network. Besides the common parts, a GET request contains the name of an NDO. The NI URI scheme must be supported. A node receiving a GET request should either generate a GET-RESP or forward the request and then generate a GET-RESP after waiting a reasonable time for response.

A GET-RESP should, if possible, contain the binary data of the NDO. If this is not possible the GET-RESP may contain locators to where the NDO is available. When building on top of current networks, a locator could for example be an IP address or a Bluetooth MAC address. The GET-RESP may also contain metadata about the NDO as well as other extensions.

The name-data integrity allows the receiver to verify that the received binary data is what was actually requested. This is simply done by applying the digest algorithm from the requested NI URI to the received data, and verifying that it produces the same digest value as in the requested NI URI. The name-data integrity is important. In a NetInf network the requester has no control over from where the response is sent, and needs to be able to trust the received data.
PUBLISH/PUBLISH-RESP

A PUBLISH request is sent in order to tell a node that an NDO exists. The PUBLISH must contain an NDO name. It may also include locators, alternative names, the binary data, and/or metadata. A node receiving a PUBLISH request decides what to save. If it receives binary data it should most likely verify the name-data integrity. If the receiver already has information about the NDO in the PUBLISH request, the data should be merged or updated.

A PUBLISH-RESP may contain metadata, such as the time of publish or locators to additional cached copies generated by the publish, as well as other extensions.

SEARCH/SEARCH-RESP

A SEARCH request is sent in order to find the name of an NDO. The SEARCH request contains a number of tokens, keywords that describe what is being searched for. If the searcher is interested in videos featuring rainbows and unicorns, the tokens might be “video”, “rainbow”, and “unicorn”. A SEARCH request may be forwarded, as with GET requests.

A SEARCH-RESP contains a list of NDO names that are considered to match the tokens. Metadata corresponding to each match can help the searcher to choose between the results. This could for example be timestamps or information about which keywords matched.

2.2.3 Convergence Layers

CLs enables communication between NetInf nodes. There are currently two CLs being worked on [2, 3], the HTTP CL and the UDP CL. As the names indicate these transfer NetInf messages using HTTP and UDP respectively. In Chapter 4, a new Bluetooth CL is discussed. A CL must at least fulfill the following requirements:

1. One-way delivery of NetInf messages.
2. Preservation of NetInf message boundaries.
3. Reliable transmission of messages.
4. In-order delivery of binary data (within a given message).
It is up to the CL to provide these features if the protocol the CL is built upon does not provide them.

2.3 OpenNetInf

OpenNetInf [7] is an open source implementation of NetInf in Java [23]. It was developed at the University of Paderborn, using the NetInf architecture that arose during the 4WARD project. The development of OpenNetInf seems to have been stagnant since 2011 when the last source code and documentation updates were published.

2.4 Android NetInf Prototype

The existing Android NetInf Prototype uses parts of OpenNetInf. The prototype was developed during a previous master’s thesis [5] and extended in a project course [6]. While the prototype provides the basic functionality needed for using NetInf on Android it has a few drawbacks.

First, in several places it only provides the minimal, or overspecialized functionality, to make the applications developed in parallel with it work. For example, the database is overspecialized, requiring information that should not be mandatory. Moreover, while sending files over Bluetooth is supported, it is not done using a CL and hence does not provide the expected NetInf functionality.

Secondly, while the usage of OpenNetInf has influenced the design of the prototype in a good way, it actually uses very little of the OpenNetInf code. In essence only three components are used: The NDO representation, as well as the Resolution Controller and Search Controller, respectively responsible for controlling the execution of GET/PUBLISH and SEARCH requests.

Finally, the prototype suffers from randomly being killed by the Android memory management to free up memory, as it is running as a separate application in the background.

Because of the reasons stated an (almost) complete rewrite of the prototype as an Android Library without the OpenNetInf dependency is performed. Since only a few parts of OpenNetInf were used, this means there are less things to take into consideration when making changes. This allows for
quicker changes, which is important at the development stage. The rewrite learns from OpenNetInf and the prototype, using a similar design while trying to be more open for future extensions. Because of the implementation as a library the memory management is a non-problem as the library will not be killed unless the application using it is killed.
Chapter 3

Android NetInf Library

3.1 Design

The Android NetInf Library uses a design similar to that of OpenNetInf, and hence the previous prototype. Figure 3.1 shows the basic idea behind the design.

![Diagram of Android NetInf Library node design]

Figure 3.1: Abstract overview of the Android NetInf Library node design.
The responsibility of the inputs are to receive requests and return responses using some CL. When an input receives a request, it first translates it into an internal representation. The internal representation is then handed to the controller. The controller decides which outputs to use to resolve the request, and passes the request on to them. The responsibility of each output is to, given a request produce a response. This can be done by forwarding the request to another node over some CL and then wait for the response, or by performing a lookup in a local database. This means that an outgoing request from an output can become an incoming request to an input on another node. Either way, it creates an internal representation of the response and passes it back to the controller. The controller then passes it back to the receiving input, which sends it back to the requester using its CL. Typically this starts with a local request generated through Java method calls. The request does a number of hops between nodes over a CL before being found in the local database of some node. The response then travels back using the reverse path.

Since all requests passes through the controller this is where repeated requests are dropped and request aggregation takes place. Repeated request can be detected by keeping track of the msg-id of all outstanding requests. Request aggregation can be performed if multiple GET requests with different msg-ids are received for the same NDO. In this case the GET-RESP belonging to the first GET can be used for all the subsequent requests, resulting in a reduced number of outgoing request and hence reduced bandwidth usage.

The separation into inputs, control logic, and outputs tries to ease future extensions. Support for new CLs can be added as inputs and/or outputs, depending on if messages are to be received, sent, or both. The control logic can be reused for such extensions as it is separate and uses a common internal representation.

### 3.2 Implementation

The Android NetInf Library tries to follow the Code Style Guidelines for Contributors [24]. Among a lot of other things, this means the that non-public, non-static class members are prefixed with “m”, as can be seen in the upcoming class diagrams.

The design discussed in Section 3.1 is implemented as an Android library using Java. Before describing the implementation of inputs, outputs and
control logic in Section 3.2.3, the NDO and message representations are discussed in Sections 3.2.1 and 3.2.2.

### 3.2.1 Ndo

The NDO representation is a fundamental part of the Android NetInf Library. Figure 3.2 shows a simplified view of the classes making up the NDO representation, only the relevant fields and methods are shown.

![Diagram of Ndo classes](image)

Figure 3.2: Simplified overview of the classes making up the NDO representation, only the relevant fields and methods are shown.

The main class of the representation is `Ndo`. The fields `mAuthority`, `mAlgorithm`, and `mHash` respectively stores the authority, digest algorithm, and digest value of the NDO’s NI URI. The field `mOctets` may contain the octets of the NDO if they are available. The `Ndo` also contains a `Metadata` object storing any metadata about the NDO it represents. `Metadata` internally stores the metadata as a `JSONObject`, a class used to represent JSON objects. This representation was chosen since both the
HTTP CL and UDP CL use JSON to represent metadata in their NetInf protocol messages, easing the generation of these messages. Finally, the Ndo also contains a Locator set representing potential locators to the NDO.

The Android NetInf Library is parallel in nature. Because of this, the Ndo class was made immutable (which means its fields can only be read, not changed). This adds some overhead, but greatly eases the implementation, since a thread can be certain that an Ndo object never will be changed by another thread. Since the Ndo class is immutable and has several optional fields (mAuthority, mOctets, mMetadata, mLocators) a lot of different constructors are needed to accommodate all possible combinations of optional fields.

The builder pattern was chosen to avoid this problem. The Ndo class contains a static inner class called Builder that provides all the methods required to set the fields. An Ndo can then be constructed by creating a new Builder, providing any required fields in its constructor. After using any of its methods to set optional fields, calling the build() method (which in turn calls an Ndo constructor accepting a single Builder) then returns a new Ndo. Listing 3.1 shows an example of how a new Ndo can be constructed using a Builder.

```java
Ndo ndo = new Ndo.Builder(algorithm, hash)
    .octets(octets)
    .metadata(metadata)
    .build();
```

Listing 3.1: Example code for constructing a new Ndo using the Builder

### 3.2.2 Message

Another fundamental part of the Android NetInf Library is the representations of the different types of NetInf protocol messages. Figure 3.3 shows a simplified view of the classes making up the message representations, only the relevant fields and methods are shown. These classes map almost directly to the abstract protocol messages described in Section 2.2.2.

All message representations inherit from the abstract Message class. The mid field stores the msg-id, which is common to all NetInf messages.

The request representations inherit from the Request class, which provides the two fields mSource and mHopLimit. mSource indicates over which
incoming interface the Request was received. This could for example be through method calls in JAVA, via RESTlet, or over Bluetooth. mSource is used by the different controllers, see Section 3.2.3, to determine which outgoing interfaces to use. mHopLimit contains the remaining number of node hops the message is allowed to make. A value of zero means that the request should only be handled locally and not be forwarded to any remote node. Negative values could be used to indicate infinite forwarding, but is currently not supported.

Publish and Get both have an mNdo field corresponding to the NDO of a PUBLISH or GET. While the mTokens field in Search contains the tokens used in a SEARCH.

The response representations inherit from the Response class, which provides the mStatus field. NetInfStatus is an extensible enumeration representing the status code of the response. NetInfStatus will most likely need to be extended in the future, as at the time of writing it only supports status codes for “OK” and “FAILED”.

PublishResponse currently does not contain anything other than the status. The mNdo field of a GetResponse corresponds to the resulting NDO of a GET. This hopefully contains an extended locator set or the NDO octets. Search contains an Ndo set corresponding to the NDOs considered to match the SEARCH tokens.

Publish, PublishResponse, Get, GetResponse, Search, and SearchResponse are immutable for the same reason as Ndo. They are also constructed using their corresponding Builder, however in this case it may be argued that this is not necessary since they do not contain many optional fields. However, default values are provided for many of these fields, meaning that they in essence become optional. Moreover, by using the builder pattern the interface to the Android NetInf Library is kept consistent.
Figure 3.3: Simplified overview of the classes making up the internal representation of the NetInf protocol messages.

### 3.2.3 Node

Figure 3.4 shows a simplified view of the classes making up the NetInf node. One of the changes compared to OpenNetInf is that the ResolutionController
has been split into the PublishController and GetController. An additional LogController has also been introduced.

Node is a singleton class that represents the entire node as discussed in Section 3.1. It contains a PublishController, GetController, and SearchController, each implementing part of the abstract controller. They respectively handle Publish, Get, and Search requests.

Requests are submitted to the Node using their respective submit() methods (see Figure 3.3). Each of these three controllers contain mappings from inputs (represented by the Api interface, see Figure 3.5) to sets of outputs (represented by the PublishService, GetService, and SearchService interfaces, see Figure 3.6). These mappings can be provided when the Node is started using start(...) and tells the controllers which inputs are connected to which outputs. For example, all inputs should probably be connected to the output representing the local database. Submitted requests find their way to the correct controller, where the mappings are used to select outputs. The mappings are divided into local and remote. Outputs in the local mapping are always used while those in the remote mapping only are used if the request is allowed further hops.

The submit() method is asynchronous and returns a Future representing the future response. It is up to the caller to decide how long to wait for the response.
Figure 3.4: Simplified overview of the main classes making up the NetInf node.
3.2.4 PublishController

The PublishController is the simplest of the controllers. It simply calls perform() of the appropriate PublishServices. All the responses are aggregated, only returning success if all calls succeeded.

3.2.5 GetController

When the GetController receives a request it first checks if a request with the same msg-id has been received and is in progress. If so, it immediately returns a failed result to the new request as to prevent network loops.
In the future a specific return value representing “in progress” should be used instead. This functionality is provided by `InProgressTracker<K, V>`.

The `GetController` then performs request aggregation. This means that only the first request for a specific `Ndo` is let through. Subsequent requests for the same `Ndo` are stopped until the first is resolved. When the response for the first request is received it is also used as the response for the stopped requests. This prevents duplicated requests, and hence duplicated responses, from being sent. This reduces the amount of data transferred. This functionality is provided by `RequestAggregator`.

If the request is let through the request aggregation the `GetController` starts calling `perform()` of its `GetServices`, resolving `Locators` as needed using `resolveLocators()`. Local `GetServices` are used before remote ones. A successful response is returned as soon as a response containing an `Ndo` with octets is received. A failed result is returned should all `GetServices` fail.

### 3.2.6 SearchController

Unlike the `GetController`, the `SearchController` calls all its `SearchServices` in parallel. The search results in the received responses are aggregated into a single response. This response is always a success, but the set of search results might be empty.

### 3.2.7 LogController

The `LogController` centralizes logging of NetInf messages going in and out of the node. It contains a list of `LogServices`. Every log event is passed on to each `LogService`.

### 3.2.8 Api

Classes implementing the `Api` interface need to implement `start()` and `stop()`. These methods should perform any necessary setup and tear down, for example starting and stopping a Bluetooth server.
There are currently two $Api$ implementation plus one additional way for the node to receive requests. The additional way is through Java method calls, more specifically through submit() of each request type, which in turn calls submit() of the Node. In the case the mSource of the request is set to the constant $Api$.JAVA.

The first implementation is provided by RestApi. This is one of the few remaining parts from the previous prototype. It provides a RESTlet interface to the node. It was previously used as the primary way to access the node. It became practically unneeded with the addition of the Java interface. However, in the future this code could be changed to accept HTTP CL messages or be adapted to intercept HTTP requests sent by some black box application.

The second implementation is provided by BluetoothApi and accepts messages over Bluetooth, see Section 4.2.3 for details.

### 3.2.9 PublishService/GetService/SearchService

Classes implementing the PublishService, GetService, and/or SearchService interface(s) need to implement the corresponding perform(), which should generate a response given a request. Furthermore, when implementing the GetService interface, resolveLocators() must be implemented. This method should produce a GetResponse using any compatible Locators in the request.

Database implements all three of these interfaces. It uses a SQLite database to keep track of Ndos that are locally available.

HttpPublishService, HttpGetService, and HttpSearchService respectively implements the PublishService, GetService, and SearchService interfaces. They send requests using the HTTP CL.

BluetoothPublish, BluetoothGet, and BluetoothSearch also implements these interfaces. They send requests using the BCL, see Section 4.2.8.

### 3.2.10 LogService

Classes implementing the LogService interface need to implement all variants of log(). Exactly how to log things is up to the implementation.
There are currently two LogService implementations. The LogCatLogger simply logs all received messages using LogCat, the logging system provided by Android. The VisualizationService logs all received messages by sending them to a visualization server using TCP.

### 3.3 Usage Example

An Android application using the Android NetInf Library must request certain permissions. These permissions are specified in the application’s “AndroidManifest.xml” file. The reason for this is that currently the application’s manifest overwrites the library’s. The following permissions need to be requested:

- `android.permission.INTERNET`
- `android.permission.WRITE_EXTERNAL_STORAGE`
- `android.permission.BLUETOOTH`
- `android.permission.BLUETOOTH_ADMIN`

Furthermore, to be able to show the settings menu, the settings activity needs to be declared in the manifest, see Listing 3.2. The settings menu can then be shown by calling `Node.showPreferences()`.
Listing 3.2: The settings activity needs to be declared in the application’s manifest.

Listing 3.3 demonstrates how the Android NetInf Library can be used. No error checking is performed in this simple example. In reality, the statuses of responses should be checked using `getStatus()`. If the `Publish` failed for some reason the subsequent `Get` could also fail.
// Start the node using default mappings
Node.start();

// Get octets, digest algorithm, and calculate hash
byte[] octets = ...
String algorithm = "sha-256";
String hash = ...

// Create an Ndo
Ndo ndo = new Ndo.Builder(algorithm, hash)
  .octets(octets)
  .build();

// Create a Publish
Publish publish = new Publish.Builder(ndo)
  .build();

// Submit the Publish and wait for response
Future<PublishResponse> futurePubResp = publish.submit();
PublishResponse publishResponse = futurePubResp.get();

// Create a Get
Get get = new Get.Builder(ndo)
  .build();

// Submit the Get and wait for response
// Should come from the local node since we just published
Future<GetResponse> futureGetResp = get.submit();
GetResponse getResponse = futureGetResp.get();

// Get the Ndo from the response
Ndo getResponseNdo = getResponse.getNdo();

// Get the octets from the Ndo
byte[] getResponseOctets = getResponseNdo.getOctets();

Listing 3.3: Example code that submits a Publish and then retrieves the published Ndo using a Get.
Chapter 4

Bluetooth Convergence Layer

The purpose of the BCL is to facilitate the transport of NetInf requests and responses over Bluetooth. Section 4.1 describes the structure of messages, while Section 4.2 goes into the details of the Android specific implementation.

4.1 Specification

4.1.1 Assumptions

An important thing to note is that the BCL only specifies how messages should be structured. It does not provide any means of discovering or selecting which devices to route to. Solving this problem is left up to the implementation, see Section 4.2. NetInf routing is a current area of research [25].

The BCL does not provide reliable transmission which is required by CLs. Instead, it makes the assumption that this is provided by the underlying Bluetooth protocol. This could for example be radio frequency communication (RFCOMM) [26].

In essence the BCL needs a reliable byte stream to operate. This does not necessarily have to involve Bluetooth, it could for example be used over TCP.
4.1.2 Messages

The messages sent over the BCL consists of one or two parts. Each part is prefixed by the length of the part. Since these length prefixes have fixed sizes, there is a limit to how large the parts can be. In the future it might be of interest to use a scheme which allows arbitrarily sized parts, such as Multipurpose Internet Mail Extensions (MIME) [27]. However, this comes at the cost of increased message complexity.

The first part is a mandatory JavaScript Object Notation (JSON) part, representing the message. The JSON part contains the information required by the NetInf protocol [3] as well some additional information. This includes the type of message, msg-id, etc.

The second part is optional and can be used by a PUBLISH or GET-RESP. If present, this part consist of the octets belonging to the NDO.

For an in depth description of the protocol, see Appendix C.

4.2 Implementation

4.2.1 Android Bluetooth API

Android’s Bluetooth API provides the following features used by the BCL implementation:

- Detection of other nearby Bluetooth devices
- Connection to other devices using service discovery
- Reliable transmission using RFCOMM channels

Sadly, depending on physical device and Android OS version, the Android Bluetooth API has a tendency to produce unexpected results. For example, when executing the code in Listing 4.1 on the provided devices using Android 4.0.4, the final call to accept() never returns.
BluetoothServerSocket server = BluetoothAdapter.
   .listenUsingRfcommWithServiceRecord("com.example", uuid)
   ;
server.accept();
server.close();

BluetoothServerSocket server = BluetoothAdapter.
   .listenUsingRfcommWithServiceRecord("com.example", uuid)
   ;
server.accept();

Listing 4.1: When executed on Android 4.0.4 the final call to accept() never returns.

A new Bluetooth stack implementation was introduced in Android 4.2, replacing the earlier implementation based on BlueZ [28]. This new implementation is supposed to improve compatibility and reliability, but comes with new problems. For example, one common problem [29] is triggered by simply accepting connections. When the 15th connection is accepted, Bluetooth stops working until restarted.

The main action taken by the Android NetInf Library’s BCL to avoid these, and other, bugs is to reuse Bluetooth connections. This reduces the amount of Bluetooth API calls and hence the risk of failure. In worst case, Android NetInf Library can restart Bluetooth, should some error known to be fixed by this occur. This functionality is however limited, and the only real solution is to wait for newer Android versions.

4.2.2 Overview

Figure 4.1 shows a simplified overview of the classes making up the BCL implementation. BluetoothApi provides the Api implementation. BluetoothPublish, BluetoothGet, and BluetoothSearch respectively implements PublishService, GetService, and SearchService. The other classes help with setting up and managing the Bluetooth connections.
Figure 4.1: Simplified overview of the classes making up the BCL
4.2.3 BluetoothApi

When `start()` of the BluetoothApi is called two things happen. First, an instance of BluetoothDiscovery is create and setup to periodically scan for nearby Bluetooth devices. Secondly, an instance of BluetoothServer is created and starts to listen for incoming Bluetooth connections.

4.2.4 BluetoothDiscovery

BluetoothDiscovery does not only search for devices but also allows access to them. The method `getBluetoothDevices()` returns a filtered set of devices. Which devices are filtered depends on what routing option is selected in the settings. Four routing schemes are available. Routing can be done to all or none of the available devices. It can also be done to all bonded devices or to a static list of devices.

4.2.5 BluetoothServer

BluetoothServer listens for incoming connection requests. This is done using the Android Bluetooth API. The connection is established using service discovery protocol (SDP) [30] with the Universally Unique IDentifier (UUID) [31] 111a8500-6ae2-11e2-bcfd-0800200c9a66 and uses RFCOMM [26].

Each accepted connection is added to the BluetoothSocketManager.

4.2.6 BluetoothSocketManager

BluetoothSocketManager keeps track of all open Bluetooth connections. Opened connections are added, and closed ones removed. Whenever BluetoothPublish, BluetoothGet, or BluetoothSearch need to send a request to another device, they try to get an already opened connection to that device from the BluetoothSocketManager. New connections are opened as necessary. Connections are reused to save time. Opening a new Bluetooth connection takes several seconds, see Section 6.2.1.

Whenever a connection is added or opened, a BluetoothSocketHandler is created to handle it.
4.2.7 BluetoothSocketHandler

Each BluetoothSocketHandler is associated with a connection, and is responsible for reading all incoming BCL messages and translating them into the internal representation. Requests are submitted to the local node and the response then sent back using the same connection. Responses received are given to the BluetoothSocketManager, which in turn passes it to any requests waiting for the given response.

4.2.8 BluetoothPublish/Get/Search

BluetoothPublish, BluetoothGet, and BluetoothSearch respectively implement the PublishService, GetService, and SearchService interfaces. They are responsible for generating the outgoing BCL requests of the corresponding types. BluetoothSocketManager is used to get open connections and to retrieve responses.
Chapter 5

Live Video Streaming

Live video can be streamed over NetInf using an approach similar to that of HTTP Live Streaming (HLS) [32] and Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [33]. On the recording side, a camera generates a potentially unbounded bit stream. This bit stream is chopped up into independent chunk NDOs. The viewer requests relevant chunks and play them sequentially. When watching a live stream, the playback should start at the most recent chunk. To make this possible, a manifest NDO representing the stream is created and updated for each new chunk. It contains information about the name of the latest chunk.

5.1 Dynamic Content

Ideally, each chunk $c_i$ would contain the name of the next chunk $c_{i+1}$ in its associated octets, protecting it using the name-data integrity. However, since the names discussed earlier include the digest value, this means that the name of $c_i$ would depend on the name of $c_{i+1}$. This applies to every chunk, and hence the name of the last chunk must be known before it is possible to calculate the name of any other chunk in the stream. This does not allow for live streaming, since the entire stream must be available before chunk generation can start.

One alternative is to include the name of $c_{i+1}$ in the metadata of $c_i$. In this case, the name of $c_i$ does not depend on the name of $c_{i+1}$, meaning only the name of $c_{i+1}$ needs to be known before $c_i$ can be published. However, the metadata is not protected by the name-data integrity, allowing for a potential
attack by changing the name in the metadata.

The manifest also suffer from a similar problem. Ideally, the name of the stream, i.e. the manifest, would be constant. This makes it easy to reference the stream even though it is dynamic in nature. For this to be possible the name of the manifest can not be directly dependent on the octets, i.e. it can not include the digest value.

One way to solve the name-data integrity problems introduced by dynamic content is to use digital signatures [34]. In simple terms, this works by the publisher having a private and a public key. The private key is known only to the publisher while the public key, as the name indicates, is public knowledge. The octets intended to be published are extended with a signature generated using the octets and the private key. The hash of the public key is included in the name of the resulting NDO together with some arbitrary name for the object. This name can be generated in advance, since it does not depend on the octets. Using the name of the NDO, the public key and the signature, it is then possible to verify that the octets have not been changed since they were published and by whom they were published. This provides a type of name-data integrity, but requires trust in a verifiable publisher.

There are at least two different solutions. The first solution, Figure 5.1, is to let all chunks have ordinary names and include all of them in the manifest as they become available. In this case, only the manifest needs to be signed. The second solution, Figure 5.2, is to sign each chunk and link them together.

Figure 5.1: A solution to the dynamic content problems related to live video streaming. The chunks use ordinary names, while the manifest is signed.
Figure 5.2: Another solution to the dynamic content problems related to live video streaming. Both the manifest and the chunks are signed.

Unless it is acceptable to miss chunks, the first solution requires that all chunk names are stored in the manifest. It has the advantage that the processing power used for signing can be easily controlled. This is done by adding multiple chunks to the manifest at the same time. However, doing so introduces a delay into the stream. It also suffers from the drawback that the manifest needs to be repeatedly downloaded to get the name of the next chunk, which increases the network traffic.

The second approach requires more processing power, as each chunk needs to be signed. The processing power allocated to signing can be decreased by signing groups of chunks [35]. This also introduces a delay into the stream. The linking can be explicit, including the name of the next chunk in the octets, or implicit, by for example including a chunk number in the name.

Combinations of these solutions, or entirely different ones, are of course also possible. For example, the manifest could contain information about all chunks even though the chunks are signed and linked. This would allow playback to be started from any point of the stream.

5.2 Live Video Streaming Application

The Live Video Streaming Application (LVSA) is an proof of concept Android application that demonstrates the usage of the Android NetInf Library by enabling live streaming of video. For the purpose of this thesis, an implementation using digital signature is out of scope. Instead, the LVSA violates the name-data integrity, being content with the fact that the problem could be solved using digital signatures as discussed in Section 5.1.

The LVSA names the manifest “ni:///video;name” where “name” is an arbitrary name chosen for the stream. Chunks are named “ni:///chunk;name-
Table 5.1:

<table>
<thead>
<tr>
<th>NDO</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manifest</td>
<td>ni:///video;news</td>
</tr>
<tr>
<td>Chunk 0</td>
<td>ni:///chunk;news-0</td>
</tr>
<tr>
<td>Chunk 1</td>
<td>ni:///chunk;news-1</td>
</tr>
<tr>
<td>Chunk 2</td>
<td>ni:///chunk;news-2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

number” where “name” is the same as for the manifest and number is the chunk number. Table 5.1 shows what the NDO names would look like using the stream name “news”.

This naming scheme is used since the name of any chunk can be easily calculated. Given the manifest name, simply change the algorithm to “chunk” and append the chunk number.

Advanced Video Coding (AVC) [36] is used to encode the video. It was chosen since it is a widely used format and easy to split into chunks. For simplicity’s sake, the LVSA only supports a single encoder. This encoder, the Texas Instruments Ducati H.264 encoder, is available on the Samsung Galaxy Nexus devices used during development. This only places a restriction on which devices can generate the stream, not on which can watch it. This is an area of future work.

In this case, the encoded video starts with a block that contains information about the stream, e.g. resolution, and is followed by a possibly unbounded number of blocks containing video. The video blocks reference the information block, hence the information block is required to be able to watch the stream. For this reason the information block is placed on its own in chunk 0.

On the recording side, the LVSA performs the following using the Android NetInf Library:

- A camera records video.
- The video is encoded using AVC.
- The encoded video is split into the information and video chunks.
- The chunks are published.
- The manifest is published and updated for each chunk.
On the watching side, the LVSA performs the following using the Android NetInf Library:

- The manifest is requested.
- Chunk 0, containing the stream information is requested.
- The current chunk, as specified by the manifest, is requested.
- Subsequent chunks are requested.
- Retrieved chunks are merged into a single video file.
- The video file is played using the VLC media player [37].

Should a request result in a failed response, the LVSA simply waits a little while before resending the request.

One thing to keep in mind when updating NDOs is the potential caching problems. When requesting the manifest file, only the most recent version is interesting. Cached copies would cause the stream to start from the wrong chunk. In the LVSA this is simply solved by not caching the manifest. A future extension to NetInf could allow for an expiration time to be specified for cached NDOs.
Chapter 6

Evaluation and Conclusions

6.1 Measurements

To provide some insight into under what conditions it is reasonable to use the BCL, the transfer time of different sized NDOs is measured. The time needed to transfer an NDO of a given size between two devices are shown in Figure 6.1. From these measurements the transfer rates of different sized NDOs are derived, see Figure 6.2. An NDO size of zero represents a GET that did not find the requested NDO. The full measurement data is listed in Appendix B.
Figure 6.1: Average time needed to retrieve an NDO using a GET over the BCL, both when establishing a new connection and when reusing an already established connection.
Figure 6.2: Average transfer rate when retrieving an NDO using a GET over the BCL, both when establishing a new connection and when reusing an already established connection.

For comparison, Table 6.1 shows the time needed to return a failed response when only looking in the local database.

<table>
<thead>
<tr>
<th>Device</th>
<th>Average Response Times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Galaxy Nexus</td>
<td>15 12 15 14 15 14 17 18 18 15</td>
</tr>
<tr>
<td>HTC One X</td>
<td>12 30 12 15 9 8 7 8 9 8 9 9</td>
</tr>
<tr>
<td>HTC One X+</td>
<td>9 12 8 8 8 8 8 8 14 11 8 8</td>
</tr>
</tbody>
</table>

Table 6.1: Time needed to perform a GET, resulting in no octets, in the local database.
6.2 Discussion

6.2.1 Bluetooth Convergence Layer

Comparing the transfer times when using a new connection and when reusing an existing one shows that the time required to setup a Bluetooth connection is in the order of seconds, see Figure 6.1. This is why the BCL keeps track of open connections and reuses them. By reusing existing connections the setup time is, in the best case, a one time cost. Of course the connection might be closed for some reason outside of the library’s control, in which case the connection needs to be reopened.

Figure 6.1 also show that the time needed to generate the response to a failed GET when using the BCL is, depending on phone, approximately 50-150 ms. This is the time elapsed from when the GET is submitted to the Android NetInf Library, until the GET-RESP is returned. Of this, roughly 10-15 is spent on the local database lookup, see Table 6.1. It is important to note that this is the time required for a single node jump. Further jumps will increase the time linearly as each forwarded GET needs to be finished before the response can be returned.

Figure 6.2 shows that the transfer rate decreases as NDO size decreases. The reason for this is that the overhead caused by the library is relatively large when transferring small NDOs. As NDO size and hence the time needed to transfer the octets increases the transfer rate levels out, approaching the limit imposed by the available Bluetooth bandwidth. The devices used in this test reached roughly 190-200 KIB/s. It is important to note that this is the ideal transfer speed, which assumes that the NDO name is known and available from the first queried device. Bluetooth can be an alternative depending on the current quality of other transfer methods such as 3G. It can be used as a substitute or a complement to reduce congestion. Flash crowd events is one possible such scenario. Another is anticipated high concentrations of users, such as on trains or in stadiums.

There exists a high speed version of Bluetooth which supports transfer rates of up to 24 Mbit/s (3 MB/s). However, this was not supported by any of the devices used for the test. This, or the use of other technologies such as Wi-Fi Direct, could improve the feasibility.

Figure 6.2 shows that for new connections, the transfer rate is still increasing as NDO sizes enter the MIB range. For reused connections the transfer rate increase starts to stagnate at NDO sizes of 100-200 KIB. This needs to be
taken into consideration when reasoning about the feasibility of the BCL. Depending on the data, the transfer rate might be well below the optimal. For example, at the time of writing, the front page of popular Swedish tabloid Aftonbladet (http://www.aftonbladet.se/) consists of 471 files (for a total of \( \sim 5.3 \) MIB). Of these, 462 files (for a total of \( \sim 3.5 \) MIB) have a size of less than 100 KIB. The exact composition of course depends on the web page and can vary over time. In this case, if each file was mapped to an NDO, over half the data would be transferred using suboptimal speeds. A possible solution to this is to group files together. For example, the front page could be represented by a single NDO of size \( \sim 5.3 \) MIB.

### 6.2.2 Live Video Streaming Application

It is observed that when using the LVSA the delay from recording to playback is in the magnitude of seconds. Because the application runs on different devices, a more detailed analysis would require some kind synchronization. The fact that the received video file is simply passed to VLC, further complicates this matter. If this is a problem depends on the application. In face to face communication a several seconds long delay could be devastating. On the other hand, in streamed live TV it might be acceptable. One major contributor to the delay is the fact that the application downloads a number of chunks before simply passing them to VLC. A more sophisticated implementation with control over video playback could immediately start the playback, and then pause it as necessary if the end of the current data is reached.

During the development a few different bandwidth reducing features are investigated. These are in-network caching, Request Aggregation (RA), and Conditionally Delayed Responses (CDR). The Android NetInf Library supports in-network caching and RA. CDR can be used when a node receives a GET for an uncached NDO that it expects soon will become available. In this case it can delay the response until the NDO actually becomes available, removing the need for the requester to issue another request. This is for example the case in the live streaming scenario. The recording node knows that the next chunk soon will become available. In the live streaming case a combination of in-network caching, RA, and CDR can potentially remove all unnecessary requests and duplicated responses [38]. Support for CDR is future work.
Chapter 7

Future Work

7.1 Android NetInf Library

Currently each application using the Android NetInf Library needs to have its own instance. This can cause problems such as multiple instances of inputs being started and unnecessary memory usage. In the short term, it would be a good idea to investigate methods of sharing a single instance of the library between applications, perhaps as an Android service. In the long term, operating system support for NetInf would be optimal.

A possible extension to the Android NetInf Library is the implementation of cut through forwarding. The idea is that when the GET-RESP to a forwarded GET is received, the node can start sending back the response before it is entirely received. This reduces delays, but since the whole NDO is not available, the name-data integrity can not be checked by the forwarding node before responding.

Another possible extension includes CDR, as mentioned in Section 6.2.2. This could reduce the amount of repeated requests in the NetInf network.

A more extensive change would be to rewrite the Android NetInf Library as a standard Java library, without any Android specific dependencies. This would allow the library to be used both on Android, perhaps wrapped in an Android Library, as well as on any other device capable of running Java. This rewrite would include at least three larger changes. JSON support would need to be provided, as the current implementation uses that provided by Android. Much of the logging would need to be changed, as it uses the logging system provided by Android. Finally, the database would need to be
changed, as that also uses functionality provided by Android.

Routing and dynamic content are two areas of future work. This does not only concern the Android NetInf Library, but NetInf in general. These problems need to be solved for NetInf to be successful. For example, a versioning system for dynamic NDOs which allows the lastest version to be requested, or the ability to specify expiration times for NDOs would be useful.

7.2 Bluetooth Convergence Layer

As mentioned in Section 4.1.2, the specification should in the future be altered to allow messages of arbitrary size.

Since both the Android NetInf Library and BCL specification are written in a general way, it might be useful to investigate alternatives to Bluetooth. Better bandwidth, range, and battery usage are features to take into consideration. Wi-Fi Direct is one possible alternative available today. In the future other alternatives might become available.

7.3 Live Video Streaming Application

In its current state the LVSA is a very basic proof of concept prototype. As mentioned in Section 5.2 only one encoder is currently supported. This needs to be changed if the LVSA it to be used on different types of devices. Furthermore, currently audio is not supported. For commercial application features such as encryption and digital signing of dynamic NDOs might be relevant.
References


Appendix A

Terminology

**AVC** Advanced Video Coding.

**BCL** Bluetooth Convergence Layer.

**CCN** Content-Centric Networking.

**CDR** Conditionally Delayed Responses.

**CL** Convergence Layer.

**DONA** Data-Oriented Network Architecture.

**GET** Request used to retrieve an NDO from a NetInf network.

**GET-RESP** The response to a GET.

**HCN** Host-Centric Networking.

**HLS** HTTP Live Streaming.

**ICN** Information-Centric Networking.

**JSON** JavaScript Object Notation.

**locator** A pointer to where an NDO could be available. E.g. an IP address or a Bluetooth MAC address.

**LVSA** Live Video Streaming Application.
**metadata** Extra information about an NDO not protected by the name-data integrity. E.g. the Content-Type of the NDO’s octets.

**MIME** Multipurpose Internet Mail Extensions.

**MPEG-DASH** Dynamic Adaptive Streaming over HTTP.

**msg-id** message identifier.

**NDO** Named Data Object.

**NetInf** Network of Information.

**NI** Named Information.

**PSIRP** Publish-Subscribe Internet Routing Paradigm.

**PUBLISH** Request used to announce the existence of an NDO to a NetInf network.

**PUBLISH-RESP** The response to a PUBLISH.

**RA** Request Aggregation.

**RFCOMM** radion frequency communication.

**SDP** service discovery protocol.

**SEARCH** Request used to search for an NDO in a NetInf network.

**SEARCH-RESP** The response to a SEARCH.

**URI** Uniform Resource Identifier.

**UUID** Universally Unique IDentifier.
Appendix B

Measurements
<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>897 1290 1343 767 1290 6932 1100 745 1345 7207</td>
</tr>
<tr>
<td>10</td>
<td>1622 698 7199 1494 1156 708 1280 7117 567 1574</td>
</tr>
<tr>
<td>20</td>
<td>862 1261 905 1047 573 933 1029 528 2272 1925</td>
</tr>
<tr>
<td>50</td>
<td>1492 1447 944 1309 1760 1373 1696 1376 887 1218</td>
</tr>
<tr>
<td>100</td>
<td>2095 1984 1628 2247 2044 1109 7996 2080 1433 1683</td>
</tr>
<tr>
<td>200</td>
<td>2749 3048 2696 1763 1521 2215 4282 2473 8942 1935</td>
</tr>
<tr>
<td>500</td>
<td>3907 4287 3590 4479 3548 9914 3177 3625 3182 4192</td>
</tr>
<tr>
<td>1000</td>
<td>5821 5729 7206 7253 6158 6618 6155 6162 7867 6809</td>
</tr>
<tr>
<td>2000</td>
<td>12532 12029 17542 11775 12205 13088 13885 11567 11861 12525</td>
</tr>
</tbody>
</table>

Table B.1: BCL transfer times between two Samsung Galaxy Nexus devices using a new connection.

<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>91 81 61 100 76 77 84 75 71 67</td>
</tr>
<tr>
<td>10</td>
<td>158 137 122 129 137 220 129 118 107 112</td>
</tr>
<tr>
<td>20</td>
<td>170 175 153 153 289 242 167 150 155 167</td>
</tr>
<tr>
<td>50</td>
<td>1061 387 278 287 291 295 401 298 290 457</td>
</tr>
<tr>
<td>100</td>
<td>611 585 544 595 516 803 538 598 614 533</td>
</tr>
<tr>
<td>200</td>
<td>1134 1094 1094 1107 1084 1247 1072 1147 945 941</td>
</tr>
<tr>
<td>500</td>
<td>2937 3467 2823 2829 2613 2671 2485 2535 2556 2764</td>
</tr>
<tr>
<td>1000</td>
<td>5205 5280 5243 5509 6870 5145 5394 5719 6124 5246</td>
</tr>
<tr>
<td>2000</td>
<td>10599 10147 10328 10269 11207 10877 10554 10454 11242 10624</td>
</tr>
</tbody>
</table>

Table B.2: BCL transfer times between two Samsung Galaxy Nexus devices reusing an existing connection.
### HTC One X (New Connection)

<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4917 3332 4698 4065 4288 4365 4344 4305 4338 4364</td>
</tr>
<tr>
<td>10</td>
<td>4429 4383 4419 4340 4334 4265 4367 4267 4316 4397</td>
</tr>
<tr>
<td>20</td>
<td>4365 4421 4375 4348 4366 4376 4374 4172 4295 4450</td>
</tr>
<tr>
<td>50</td>
<td>4431 4375 4409 4318 4486 4274 4398 4327 4270 4425</td>
</tr>
<tr>
<td>100</td>
<td>4791 4065 4505 4286 4316 4401 4230 4396 4364 4326</td>
</tr>
<tr>
<td>200</td>
<td>5071 4189 4329 4355 4444 4221 4407 4380 4216 4490</td>
</tr>
<tr>
<td>500</td>
<td>5791 6819 5906 6433 6461 6315 6835 5692 6438 5824</td>
</tr>
<tr>
<td>1000</td>
<td>8617 9106 8159 8186 9406 8460 6253 9962 8148 8837</td>
</tr>
<tr>
<td>2000</td>
<td>13967 14626 14250 12505 13021 14306 14497 13669 15030 13786</td>
</tr>
</tbody>
</table>

Table B.3: BCL transfer times between two HTC One X devices using a new connection.

### HTC One X (Reused Connection)

<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>166 99 152 132 142 177 108 177 109 182</td>
</tr>
<tr>
<td>10</td>
<td>239 209 200 229 194 226 169 230 213 207</td>
</tr>
<tr>
<td>20</td>
<td>224 194 283 149 265 274 270 239 239 262</td>
</tr>
<tr>
<td>50</td>
<td>406 345 398 390 368 396 368 413 350 375</td>
</tr>
<tr>
<td>100</td>
<td>650 653 713 713 653 622 674 644 617 676</td>
</tr>
<tr>
<td>200</td>
<td>1146 1128 1084 1461 1349 1169 1186 1081 1169 1068</td>
</tr>
<tr>
<td>500</td>
<td>2712 2624 2668 2690 2604 2706 2684 2624 2682 2685</td>
</tr>
<tr>
<td>1000</td>
<td>5141 5833 5433 5344 5258 5335 5142 5105 5170 4799</td>
</tr>
<tr>
<td>2000</td>
<td>10041 9491 10016 13458 10104 9783 9763 10201 11172 11021</td>
</tr>
</tbody>
</table>

Table B.4: BCL transfer times between two HTC One X devices reusing an existing connection.
## HTC One X+ (New Connection)

<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2225 1509 1372 631 3686 2354 3842 1583 744 2319</td>
</tr>
<tr>
<td>10</td>
<td>3779 1832 2584 1979 2282 1879 2482 3022 1860 2293</td>
</tr>
<tr>
<td>20</td>
<td>2619 2036 2228 3531 1999 2499 1774 2916 1318 2087</td>
</tr>
<tr>
<td>50</td>
<td>2437 2223 2587 1957 2581 1956 2768 1705 3772 2540</td>
</tr>
<tr>
<td>100</td>
<td>2114 2823 2489 2900 5351 2232 2306 3140 2553 2765</td>
</tr>
<tr>
<td>200</td>
<td>3159 2695 2943 1621 2659 3781 2467 3009 3057 3635</td>
</tr>
<tr>
<td>500</td>
<td>3601 4212 3958 4024 5531 4520 4393 4825 3755 3836</td>
</tr>
<tr>
<td>1000</td>
<td>6933 7348 5859 6600 6541 6238 9524 5742 6194 7423</td>
</tr>
<tr>
<td>2000</td>
<td>10828 11284 12389 12993 12868 11008 12742 10947 13092 13569</td>
</tr>
</tbody>
</table>

Table B.5: BCL transfer times between two HTC One X+ devices using a new connection.

## HTC One X+ (Reused Connection)

<table>
<thead>
<tr>
<th>NDO (KIB)</th>
<th>Transfer times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>54 58 63 67 70 56 78 27 50 53</td>
</tr>
<tr>
<td>10</td>
<td>107 89 125 110 102 184 105 188 114 109</td>
</tr>
<tr>
<td>20</td>
<td>176 155 166 143 243 179 138 159 148 147</td>
</tr>
<tr>
<td>50</td>
<td>268 432 270 254 303 296 300 278 352 333</td>
</tr>
<tr>
<td>100</td>
<td>481 502 515 663 499 625 502 524 500 602</td>
</tr>
<tr>
<td>200</td>
<td>1113 990 980 952 947 1027 942 967 968 996</td>
</tr>
<tr>
<td>500</td>
<td>2463 2455 2375 2376 2473 2374 2416 2389 2340 2393</td>
</tr>
<tr>
<td>1000</td>
<td>4692 4617 4660 4974 5094 4704 4625 4741 4586 5033</td>
</tr>
<tr>
<td>2000</td>
<td>10970 9440 9763 9970 10825 9979 10653 9505 9661 10793</td>
</tr>
</tbody>
</table>

Table B.6: BCL transfer times between two HTC One X+ devices reusing an existing connection.
Appendix C

The NetInf Bluetooth Convergence Layer
Abstract

This document defines a Network of Information (NetInf) convergence layer intended to be used over some Bluetooth protocol providing reliable transmission. A convergence layer handles the transport of NetInf requests and responses. In this convergence layer the common NetInf messages are defined using JavaScript Object Notation (JSON) and Named Data Object (NDO) octet parts, both using length prefixes. While the convergence layer is intended to be used over Bluetooth, any underlying protocol providing reliable transmission could potentially be used.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 04, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

The NetInf Bluetooth Convergence Layer
draft-sunde-netinf-protocol-bluetooth-00
1. Introduction

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Bluetooth Convergence Layer Specification

This section specifies a NetInf Convergence Layer [I-D.kutscher-icnrg-netinf/proto] intended to be used over Bluetooth. The protocol does not provide reliable transmission. This is assumed to be provided by the underlying Bluetooth protocol, e.g. radio frequency communication (RFCOMM). The Bluetooth convergence layer (BCL) could potentially also be used on top of non-Bluetooth protocols providing the required features (e.g. TCP).

The BCL does not provide any means of discovering or selecting which devices to route to. This problem must be handled by the implementation. A Bluetooth connection needs to be established between the sender and the receiver. This connection MAY use RFCOMM [RFCOMM]. It MAY be established using the service discovery protocol [SDAP] with the Universally Unique IDentifier [RFC4122] 111a8500-6ae2-11e2-bcfd-0800200c9a66.
The response to a request SHOULD be sent back using the same connection (if two-way communication is supported) but an implementation MUST NOT assume that this always is the case.

The protocol is built around sending UTF-8 [RFC3629] encoded JSON and NDO octets through a stream providing reliable transmission. All JSON and octet parts MUST be prefixed with their length in bytes. The length prefix MUST be a 32-bit signed integer encoded in two’s complement using big endian network byte order.

The ni URI scheme [RFC6920] MUST be supported. Other URI schemes MAY be supported.

The following names are defined for JSON name/value pairs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>Message type. MUST be &quot;publish&quot;, &quot;publish-response&quot;, &quot;get&quot;, &quot;get-response&quot;, &quot;search&quot; or &quot;search-response&quot;.</td>
</tr>
<tr>
<td>msgid</td>
<td>string</td>
<td>Message identifier.</td>
</tr>
<tr>
<td>uri</td>
<td>string</td>
<td>NDO URI.</td>
</tr>
<tr>
<td>hoplimit</td>
<td>number</td>
<td>Number of times to forward this message. Assumed to be 0 if not present.</td>
</tr>
<tr>
<td>locators</td>
<td>[string]</td>
<td>Locators to the NDO.</td>
</tr>
<tr>
<td>ext</td>
<td>object</td>
<td>Metadata and future extensions. Metadata MUST be an object under the &quot;meta&quot; key if present. E.g. &quot;ext&quot;:{&quot;meta&quot;:{...}}</td>
</tr>
<tr>
<td>octets</td>
<td>true/false</td>
<td>Flag indicating if this JSON is directly followed by the NDO octets (including the length prefix). Assumed to be false if not present.</td>
</tr>
<tr>
<td>status</td>
<td>number</td>
<td>Code indicating the status of a response.</td>
</tr>
<tr>
<td>tokens</td>
<td>[string]</td>
<td>Tokens used in a SEARCH. MUST contain at least one token.</td>
</tr>
<tr>
<td>results</td>
<td>[object]</td>
<td>The results of a search. See Section 2.6.</td>
</tr>
</tbody>
</table>
2.1. PUBLISH

A PUBLISH MUST start with a JSON part (including the associated length prefix). The JSON part MUST include the following name/value pairs:

- type
- msgid
- uri

Where "type" MUST be "publish" and "uri" MUST be the URI of the NDO being published.

The JSON part MAY also include:

- hoplimit
- locators
- ext
- octets

If "octets" is present and true, the JSON part MUST be followed by the octets of the NDO (including the associated length prefix). If "octets" is not present or false, the JSON part MUST NOT be followed by the octets.

2.2. PUBLISH-RESP

A PUBLISH-RESP MUST consist of a single JSON part. The JSON part MUST include the following name/value pairs:

- type
- status
- msgid

Where "type" MUST be "publish-response".

2.3. GET
A GET MUST consist of a single JSON part. The JSON part MUST include the following name/value pairs:

- type
- msgid
- uri

Where "type" MUST be "get".

The JSON part MAY also include:

- hoplimit

### 2.4. GET-RESP

A GET-RESP MUST start with a JSON part. The JSON part MUST include the following name/value pairs:

- type
- status
- msgid
- uri

Where "type" MUST be "get-response" and "uri" MUST be the URI requested NDO.

The JSON part MAY also include:

- locators
- octets

If "octets" is present and true, the JSON part MUST be followed by the octets of the requested NDO. If "octets" is not present or false, the JSON part MUST NOT be followed by the octets.

### 2.5. SEARCH

A SEARCH MUST consist of a single JSON part. The JSON part MUST include the following name/value pairs:

- type
2.6. SEARCH-REP

A SEARCH-REP MUST consist of a single JSON part. The JSON part MUST include the following name/value pairs:

- type
- status
- msgid
- results

Where "type" MUST be "search-response".

Each object in the "results" array represents an NDO considered to match the tokens included in the corresponding SEARCH. If no matching NDOs were found "results" might be empty. The following name/value pairs are defined for the search result objects:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>string</td>
<td>The ni URI of the matching NDO. MUST be included.</td>
</tr>
<tr>
<td>metadata</td>
<td>object</td>
<td>Metadata belonging to the matching NDO. MAY be included.</td>
</tr>
</tbody>
</table>

Figure 2: Search result JSON
3. Security Considerations

As mentioned in [I-D.kutscher-icnrg-netinf-proto] requesters SHOULD attempt to limit the amount of personally identifying information for privacy reasons. Care should be taken when generating SEARCH tokens and when responding to a SEARCH.

Validation of name-data integrity is important. Consider a scenario where the connected Bluetooth devices are a number of smart phones that happen to be in close proximity. The other devices can not be trusted to send the correct data and might even try to send malicious data.

In the smart phone case battery life is also something to take into consideration. By requesting data over and over again a requester could intentionally drain the battery of the responder.

4. IANA Considerations

This document has no actions for IANA.

5. References

5.1. Normative References


5.2. Informative References


Author’s Address

Linus Sunde
Uppsala University
Uppsala
Sweden

Email: Linus.Sunde.9273@student.uu.se