



OpenNetInf Documentation Design and Implementation

Project leader: Christian Dannewitz, Matthias Herlich

Developer: Eduard Bauer, Matthias Becker, Frederic Beister, Nadine Dertmann, Răzvan Hrestic, Michael Kionka, Mario Mohr, Matthias Mühe, Deepika Murali, Felix Steffen, Sebastian Stey, Eduard Unruh, Qichao Wang, Steffen Weber

September 2011

Technical Report TR-RI-11-314

This documentation is based on document number TR-RI-10-318 and
supersedes this document.

Technical Report

Contents

1	Introduction	1
1.1	Main Goals of the OpenNetInf Project	2
1.2	What's New?	2
2	OpenNetInf Architecture and Implementation	5
2.1	Components	5
2.2	Node Structure	8
2.3	Package Structure	10
2.4	Modularity	11
3	Components in Detail	13
3.1	Data Model	13
3.1.1	Architecture	13
3.1.2	Implementation	15
3.2	Node Interface	16
3.2.1	Architecture	17
3.2.2	Implementation	17
3.2.3	Protobuf Messages	18
3.2.4	Convenience Features	19
3.3	Application Interface	19
3.3.1	RESTful HTTP	19
3.3.2	Legacy Support	21
3.4	Resolution	21
3.4.1	Resolution Controller	22
3.4.2	Resolution Service	22
3.4.3	Resolution Service Selector	29
3.4.4	Resolution Interceptor	29
3.5	Search	29
3.5.1	Overview	30
3.5.2	Class View	31
3.5.3	Operational View	33
3.5.4	SearchServiceRDF	35
3.6	BO Handling	36
3.6.1	DataObjects	36
3.6.2	Transfer	36

3.6.3	Chunking	37
3.6.4	Caching	40
3.7	Security	43
3.7.1	Integrity	47
3.7.2	Cryptography	51
3.8	Event Service	53
3.8.1	Event Service Framework	54
3.8.2	Event Service Siena	61
4	Generic Path Integration	65
4.1	Connection to GP	65
4.2	Name Resolution	66
4.3	Transfer	68
5	OpenNetInf Scenarios and Use Cases	71
5.1	Scenario 1: Data Integrity and Data Availability	71
5.1.1	Scenario Description	71
5.1.2	Applications	74
5.1.3	How to Run the Scenario	74
5.2	Scenario 2: Update Awareness	74
5.2.1	Scenario Description	75
5.2.2	Applications	76
5.2.3	How to Run the Scenario	78
5.3	Scenario 3: Chunking, Streaming, Caching and the MDHT	80
5.3.1	Scenario Description	80
5.3.2	How to Run the Scenario	82
6	How to	83
6.1	How to Get, Compile and Run NetInf Components	83
6.1.1	Required Libraries	84
6.1.2	System Requirements	84
6.1.3	Using Eclipse	84
6.1.4	Using Ant to Build	86
6.1.5	Command Line Operations	87
6.1.6	Known Problems	87
6.2	How to Create a Customized NetInf Node	88
6.2.1	The AbstractNodeModule Module	88
6.2.2	The Subclass of AbstractNodeModule	89
6.3	How to Modify, Develop New Components	89
6.4	How to Develop NetInf Enabled Applications	92
6.4.1	Use NetInf as Data Storage	92
6.4.2	Use NetInf Search	94
6.4.3	Get Informed by Event Service	96
6.5	How to Send and Receive Messages Using Protobuf	98
6.6	Logging	101

6.7 How to install Mozilla Extensions	102
7 OpenNetInf Applications	103
7.1 InFox: OpenNetInf Extension for Mozilla Firefox	103
7.1.1 IdentityObjects	103
7.1.2 DataObjects	104
7.1.3 InformationObjects	104
7.1.4 InFox Extension Preferences	106
7.1.5 Developing InFox	106
7.1.6 How to Generate a XPI-Package	107
7.2 InBird: OpenNetInf Extension for Mozilla Thunderbird	108
7.3 Management Tool	108
8 Open Issues	111
8.1 Extending Existing Concepts	111
8.1.1 Data Model	111
8.1.2 Security: How to Create an InformationObject That Never Existed	114
8.1.3 Actuality of InformationObjects	115
8.2 Further Concepts	117
8.2.1 Rights Management	117
8.2.2 Trust: Public Key Infrastructure	120
8.2.3 Versioning	121
8.2.4 Collaborative Caching	126
Acronyms	127
Bibliography	129
Appendix	131
A Data Model – InformationObject Ontology	133
Imprint	143

Chapter 1

Introduction

Today's computer networks use a host-based naming scheme referencing data by its location. This requires users who are interested in the data to know the location, which – in most cases – is of no use to them. The users are interested in the data, no matter where it comes from, as long as it is the data that was requested. Thus, an information-centric approach may be more suitable.

The documentation at hand describes the *OpenNetInf* prototype of the Network of Information ([NetInf](#)) architecture. The prototype is closely related to the [NetInf](#) architecture as described in the 4WARD project [1] and builds upon those concepts.

[NetInf](#) handles data independently from its location. Information about a certain topic is contained within an [InformationObject \(IO\)](#) – which represents a unit of information. Every [IO](#) has a globally unique identifier. This identifier is used to reference the [IO](#), a location in form of a host-based address used in today's Internet is not needed. Thus, the [IO](#) may be stored anywhere in [NetInf](#).

To guarantee that an [IO](#) always contains the same data, unauthorized manipulations have to be prevented. Since users can always manipulate data they have on their own computer, manipulation has to be made visible to the receiver instead. Thus, [IOs](#) have to contain information which allows anyone to validate their correctness.

As there is no need for hosts to be trusted, it is possible to arbitrarily distribute the [IOs](#) among the nodes in the network. This allows popular [IOs](#) to be stored at nodes close to emerging interest, lets the network scale with an increasing number of users, and it allows important [IOs](#) to be distributed evenly among the network to increase their availability even in case of failing nodes. Additionally, users do not necessarily have to provide the storage in the network to publish [IOs](#), since others may share their available storage to enable a greater accumulation of information.

The uniform, structured representation of data in the form of [IOs](#) has several benefits: First, [IOs](#) are specified precisely enough to also allow automatically searching through their structure in an uniform way. Second, users may also get notified when data is changed, without having to frequently search or poll. Third, different applications can benefit from synergistic effects through similar structures of their [IOs](#), since they may share the data with little effort.

This documentation explains the concepts, the architecture and a prototype implementation of such a network and its components. The architecture is completely modu-

lar, giving each node the freedom to decide what services to offer. The prototype works as a middleware connecting nodes to a network and providing all functions necessary to publish, distribute, search and retrieve IOs as well as for subscribing for and receiving events. All data classes and mechanisms required by applications to use NetInf are available as an includable library. Some example applications for test cases are presented to show possible applications and to demonstrate the advantages of NetInf for respective applications architectures. Except for the Generic Path (GP) integration discussed in Chapter 4 which is limited to Linux systems, the whole prototype is executable on Linux as well as on Windows systems. The Application Interface described in Section 3.3 is the basis for NetInf client applications.

1.1 Main Goals of the OpenNetInf Project

The main goal of the OpenNetInf Project is to develop and evaluate a communication architecture for the Network of Information (NetInf) based on such an information-centric communication paradigm. NetInf should not only provide large-scale information dissemination, but also accommodate non-dissemination applications, including interpersonal communication; it shall inherently support mobile and multi-access devices, capitalizing on their resources (for instance, storage). NetInf shall make it easy and efficient to access InformationObjects without having to be concerned with or hampered by underlying transport technologies. NetInf shall also provide links between the physical and the digital world. The architecture shall offer an interface to an application-neutral communication abstraction based on InformationObjects. The information model shall make it possible to manage these objects in a secure fashion.

The architecture shall be designed in such a way that a NetInf system is largely self-managing. The objective of this deliverable is to provide the first comprehensive OpenNetInf Architecture and Implementation, Components, and proposed information model, along with preliminary evaluation results.

1.2 What's New?

The OpenNetInf Project has been released as Open Source for one year now and a lot of things have been developed and improved during that time. To get a short overview of the changed and new parts here is a short list to help out:

MDHT A new scalable name resolution system called “Multilevel Distributed Hash Table”. It is implemented as a new Resolution Service for the Resolution Controller and combines multiple DHTs in a layer-based manner. Read more about Multilevel Distributed Hash Table (MDHT) in Section 3.4.2.1.

Caching In a nutshell with the development of the MDHT Resolution System, the OpenNetInf Prototype has been enhanced by two new caching components. The first one is called Peerside Cache and enables BitlevelObject (BO) caching on the client side – read more in Section 3.6.4.1. The second one is called Network

Cache and is mainly for the usage within a **MDHT** network. Thus they are kind of (In-)Network Caches and can be hosted, e.g., by providers – read more in Section [3.6.4.2](#).

RESTful API The external interface of OpenNetInf has been slightly reworked and split up conceptually in two different interfaces: the Node Interface, which is the existing TCP- and NetInfMessages-based communication between **NetInf** Nodes and described in [2.2](#), and the Application Interface. The latter is represented by the newly implemented RESTful Interface. This new interface simplifies the development of external applications and enables features such as video streaming in OpenNetInf. See more in Section [3.3](#).

InFox Plugin After the development of the RESTful Interface the InFox extension for Mozilla Firefox has been adapted to this interface – it now uses the Application Interface to request and resolve **IOs**. Furthermore the functionality of displaying different types of **IOs** has been enhanced. See more in Section [7.1](#).

Chunking and Transfer OpenNetInf is now able to handle single chunks of a given **BO**. These chunks are determined within the OpenNetInf Node and later merged by a newly developed Transfer Component called `TransferDispatcher`. See more about the improvements in the **BO** handling in Section [3.6](#).

Ivy To improve the usability of the code of OpenNetInf itself, the management of dependencies is now fully and automatically controlled by the Eclipse plugin Apache IvyDE. See more about it and how to handle it in Section [6.1.3.3](#).

DBMS The OpenNetInf Node Component got an embedded Database Management System that replaces the need for a running dedicated MySQL Server. Necessary tables are created on the fly now and the type of the DB can be easily and flexibly exchanged. See more in Section [6.1.2](#).

Chapter 2

OpenNetInf Architecture and Implementation

This chapter illustrates the overall architecture of OpenNetInf and the components developed in order to realize such an information-centric network. At first, the separate components and their interaction are described. Thereafter, an overview of the technical division into Java packages of these different high level components is given. Finally, some details about the modularity of the architecture are presented.

2.1 Components

The OpenNetInf implementation of **NetInf** distinguishes in general three kinds of components, which are runnable on their own. The following description of these different components is based on Figure 2.1. This figure shows only one of many possible assemblies and configurations of the different components.

Nodes The first and most important kind of component are nodes. The nodes can be found in the center of the mentioned figure. A node is a component that can provide two important services (the provided number of services is configurable). These services provide the capability to store and resolve **IOs** (Resolution Service (**RS**)) and search within the space of **IOs** (Search Service (**SS**)). The transfer of bitlevel data is controlled by the newly developed TransferDispatcher (**TD**) that is described in Section 3.6.2 and shall replaces the related Transfer Service (**TS**). The functionality of the Transfer Controller (**TC**) still remains in the prototype to allow the usage of **GP**. The structure and functionality of a node is described in more detail in Section 2.2.

The different nodes can communicate with each other in order to, for instance, store or retrieve **IOs**. The communication is done via either HTTP or TCP and may make use of Google Protocol Buffers¹ or XML for the serialization of the separate messages.

¹<http://code.google.com/p/protobuf/>

2 OpenNetInf Architecture and Implementation

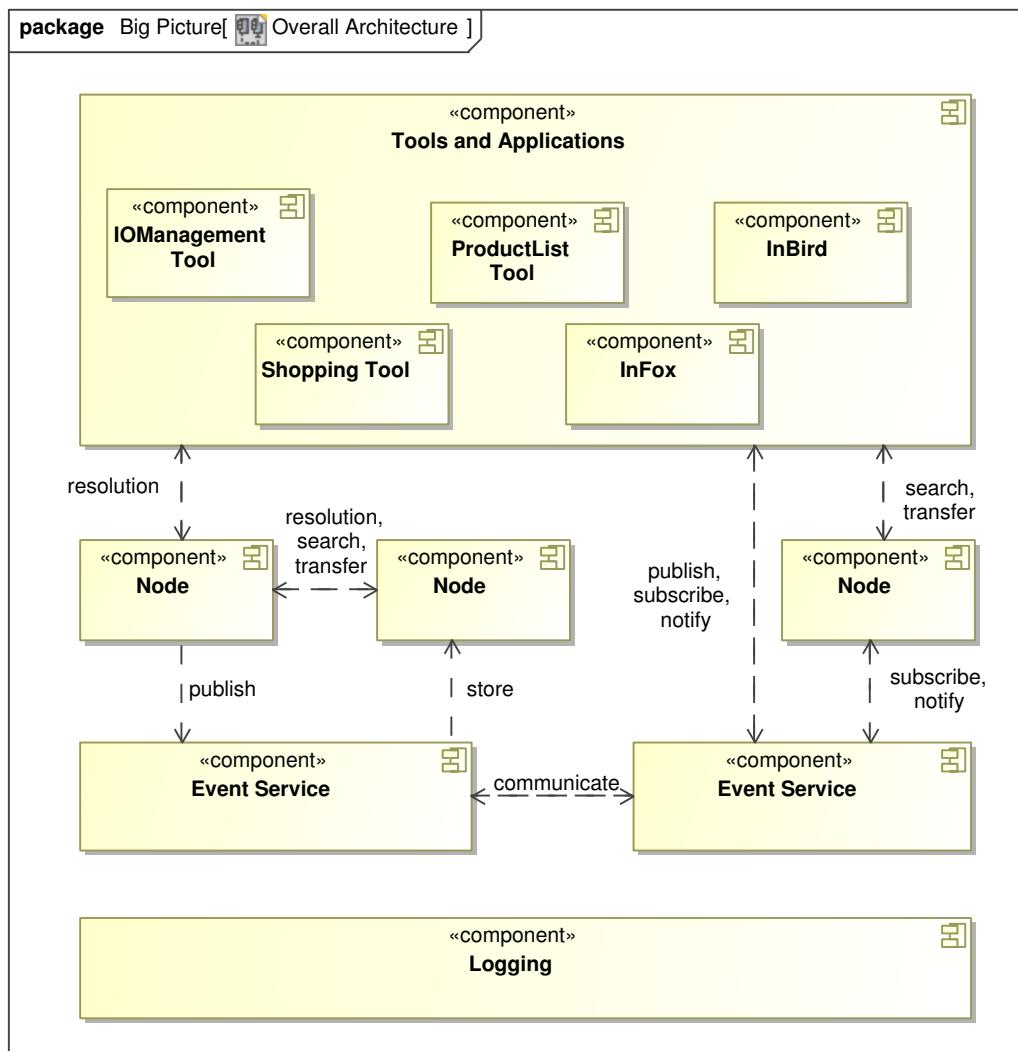


Figure 2.1: Overall architecture of NetInf

Event Service All nodes can connect to a distributed Event Service (**ES**). The components belonging to the distributed **ES** are the second kind of components. In the domain of **ES**s or publish/subscribe systems they are called event brokers. The **ES** informs about changed, created, and deleted **IOs**. For this, components have to subscribe to these events. The subscriptions are expressed in SPARQL [11]. Thus, e.g. a node that provides search for specific **IOs** might be interested in changes to these **IOs** so that it is able to keep its search index up-to-date. Accordingly, it can subscribe to the **ES** for these changes (can be seen on the right hand side of the figure). These changes are published by (other) nodes which store these **IOs** and notice an update to an **IO**. They publish these changes to event brokers, which then distribute the notification. The event service's various brokers communicate via an internal protocol.

The **ES** itself might use the services of the nodes. Consider the following case: A node subscribes to the change of a particular **IO**, and disconnects from the event broker. Now, whenever the **ES** is informed about these changes, it cannot directly inform the according subscriber, but has to store the intermediate event. In order not to create a storage service within the **ES**, we decided to provide the **ES** with the ability to store these changes (nothing else than other **IOs**) within some arbitrary nodes. This kind of communication can be seen in the center of the Figure 2.1.

Applications and Tools Finally, the third kind of components are situated on top of the node infrastructure and the **ES**. These are the applications and tools, using the advantages of the information-centric network. These components use nodes and the **ES** to provide their functionality. Several tools and applications have been developed to demonstrate the functionality of **NetInf**.

Tools and applications are allowed to use the **ES** directly and do not have to use the node to delegate their requests. The reason for this is to avoid the unnecessary overhead that would result from sending the messages via a node to the event broker. Furthermore, nodes do not belong to an application, a tool, or a particular person. But since the applications are personalized, these components have the ability to be directly connected to the **ES**. Simply forwarding the messages of applications concerning the **ES** via the nodes would accordingly result in an unnecessary overhead, and would require the functionality of the node to associate resulting messages from the **ES** to its connected applications.

Thus, for example, the ShoppingTool and the ProductListTool (described in Section 5.2.2) are separate applications, which were especially developed to demonstrate the functionality of **NetInf**. The ShoppingTool represents a client application providing a location-based service, namely the notification in case one passes a supermarket that offers a product on ones shopping list. Which products are on the shopping list can be administrated with the help of the ProductListTool.

The IOManagementTool (see Section 7.3) is necessary in order to get a detailed listing of the parts of **IOs** and how they are technically represented. This comes in handy during the development of new applications for **NetInf**. Additionally, the functionality to create, update, and delete **IOs** is incorporated into this tool.

Two other components, the InFox and the InBird, are extensions for currently ex-

isting applications, namely Firefox and Thunderbird (see Chapter 7). They provide an example how legacy applications might make use of the advantages of an information-centric network. The InFox is capable of displaying IOs and invoking conventional URLs which are stored within such IOs. It hides the technical details of fetching the IOs and evaluating the links. The aforementioned capability results in browsing information without being bothered by the location of the information and the entailed disadvantages of location changes. The InFox, as well as the InBird can readout the e-mail address of a particular kind of IO and use this e-mail address for writing e-mails. This indirection of storing the e-mail address within the IO provides the possibility to change e-mail addresses, without having to notify all people storing the e-mail address locally.

Logging Underlying all these components, one separate component exists that has a rather technical motivation. Along the development of the different (possibly distributed) components, it was necessary to develop a centralized logging server where all logging messages of the different components are collected. This provides the possibility to uncover the kind of operation and communication between the components that would have otherwise not been visible. Accordingly, each component has the possibility to send its log messages to the centralized logging server. See Section 6.6 for details.

2.2 Node Structure

Note The functionality of the TC and TS is still part of the OpenNetInf prototype, but is internally replaced by the TD. Transfer of BOs is fully controlled by the TD. Currently there is only one TS in relation to GP available. Please take this note into account when reading the description of the node below.

This section is devoted to the inner structure of one node. It should give a brief overview of the functionalities and the constituents of a node within NetInf.

Figure 2.2 contains a class diagram that depicts the most important parts of a typical node. A node can be addressed by other nodes via the NetInfServer, respectively instances of its subclasses the TCPServer and the HTTPServer. Accordingly, a node is accessible via TCP and via HTTP. The format of the messages sent to and received from a node is variable. The current implementation supports two different message representations, the first one is based on Google Protocol Buffer messages, the second one makes use of simple XML. Both formats are independent of operating system and programming language. These messages are translated into internal messages, so-called NetInfMessages. These messages are delegated by the NetInfServer to the NetInfNode. The instance of the according impl-class implementing this interface represents the entire node and is capable of processing NetInfMessages. See a more detailed description of the Node Interface in Section 3.2. In addition the NetInfNode provides a second interface that is especially developed for external applications (see Section 3.3).

As already mentioned, every node may provide three different kinds of services, which are the storing and retrieving (called resolution) of IOs, the searching within the

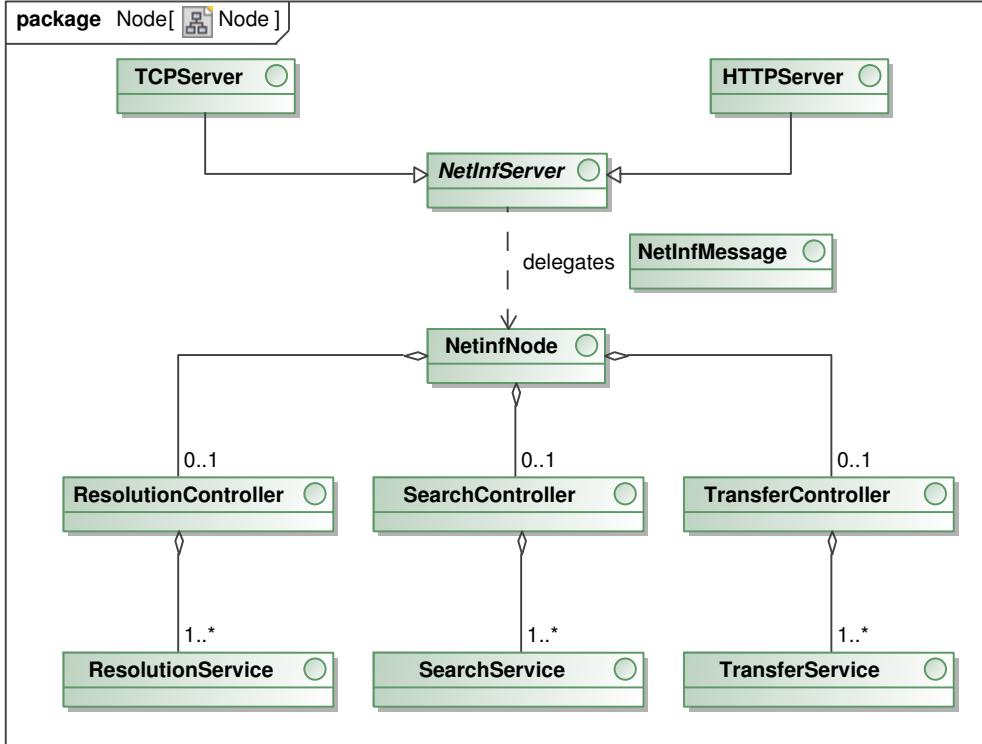


Figure 2.2: Overall architecture of a node

space of **IOs**, and, finally, the transfer of bitlevel data. Accordingly, each node uses three different controllers in charge of providing and managing these different kinds of services. For that, each of the three different controllers contains an arbitrary amount of services of the same kind. Each service in turn might achieve the intended task in a different way. This pattern of one controller managing several services is used for each of the three distinct functionalities.

The **ResolutionController** provides the capability to store, retrieve, update, and delete **IOs**. For achieving this task, the controller manages several different **RSS**, instances of classes which implement the **ResolutionService** interface. One **RS** might, for example, store the **IOs** within a global pastry ring, another might use a local database, and yet another might simply ask another node. According to different algorithms, the controller might select a service that it deems appropriate to fulfill the request.

The **SearchController** is in charge of providing the search capabilities, this means the evaluation of search requests. Each search request is represented by a simple SPARQL [11] query. Similar to the structure of the **ResolutionController** and the separate **RSS**, the **SearchController** uses several **SSs** to satisfy a search request. Each **SS** has to implement the interface **SearchService**.

Finally, the **TransferController** manages the **TS** to provide the functionality to transfer binary files, called **BOs**. Because of that, it becomes possible to copy files from remote locations to local locations and thus distribute binary files within **NetInf**.

The structure of the TransferController resembles strongly the structure of the other controllers.

These BOs are stored locally at different nodes within NetInf. The storage of BOs is described in [3.6](#).

Other Parts of the Node Besides these most basic parts of a NetInf Node, the node consists of several other parts concerning caching of IOs and binary data, security-related aspects, and different implementation-related parts. The security-related part is the most important one of these remaining parts. It is in charge of providing security mechanisms, such that no central authority is needed, in order to put trust into the data.

All of these parts are depicted in Chapter [3](#).

2.3 Package Structure

This section gives a slight introduction to the package structure, which we have used to implement the concepts explained above. The most important packages can be seen in Figure [2.3](#). The arrows express the dependency and the usage relation respectively. The package structure corresponds to the Eclipse project structure that we used during development. The explanation starts at the top.

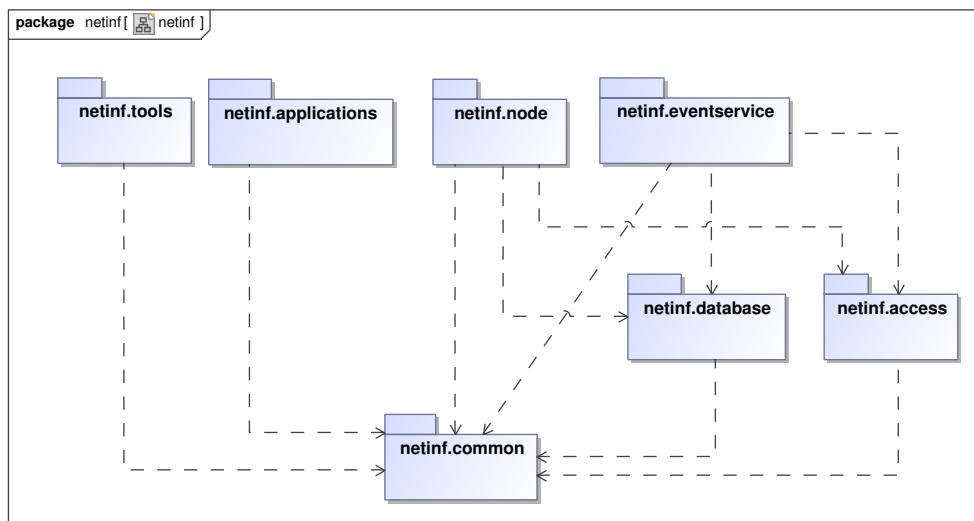


Figure 2.3: Arrangement of code into packages

Subpackages of the netinf.tools package contain tools like the afore mentioned IOManagementTool, the ShoppingTool and the ProductListTool. Additionally, the central logging server can be seen as one tool and is thus accordingly situated beneath this package. The netinf.applications package contains the InFox and the InBird. Similarly, netinf.node and netinf.eventservice contain the implementation of the node and the implementation of the ES.

All the common functionality is implemented in `netinf.common` and can be considered as one large archive that can be used to work with the Information-Centric Network. It provides a possibility to serialize **IOs** and messages and send them to nodes or event brokers easily by using several helper classes and methods. Additionally, it contains the entire implementation of security-related aspects.

Finally, the node and the **ES** both use two distinct functionalities which are implemented in separate packages. These are the possibility to access databases, implemented in the package `netinf.database`, and the possibility to provide a server to be accessible by applications, nodes and event brokers, provided by the package `netinf.access`.

2.4 Modularity

As mentioned in the previous sections, we designed a very modular architecture that, for example, allows us to define which services are offered by an instance of a **NetInf** Node. To achieve this modularity, we use dependency injection. In particular, we integrated *Guice*, a dependency injection framework for Java that is developed by Google. This section briefly describes how *Guice* is used by our **NetInf** implementation and is not meant to be a substitute for the official *Guice* documentation².

Guice provides support to bind implementations to interfaces. This functionality can be used to configure a **NetInf** Node that fulfills specific needs. For example, it is possible to configure a **NetInf** Node that has a **RS** but lacks a **SS**. And since multiple variants of **RSs** (e.g. local and distributed) have been implemented, *Guice* can be used to configure the **RSs** that a particular **NetInf** Node is supposed to use.

An implementation that has been bound to an interface will be used whenever one asks *Guice* to create an instance of this interface. Binding an implementation to an interface happens in so-called *Guice* modules. The *Guice* module that a **NetInf** Node should use can be passed to it during startup as the first argument. This argument has to be the qualified name of a Java class that extends *Guice*'s `AbstractModule` class. In case the argument is omitted, the **NetInf** Node will load a predefined standard module.

Furthermore, the **ES** and tools, mentioned in Section 2.1, use *Guice* modules. They use modules of the `netinf.common` package to access, for example, communication- and security-related functionality.

²<http://code.google.com/p/google-guice/>

Chapter 3

Components in Detail

Now that the overall architecture of [NetInf](#) has been described, the individual components shall be elaborated upon.

First, the foundation of [NetInf](#), namely its data model, is introduced in Section 3.1. Afterwards, the already mentioned `netinf.common` package – especially the part related to communications – is described in Section 3.2. This is followed by a detailed description of the three services a [NetInf](#) Node can offer: resolution (Section 3.4), search (Section 3.5) and [BO](#) handling (Section 3.6). Section 3.7 explains another integral part of [NetInf](#) – its security concept. Finally, the [ES](#) – as the second infrastructural part of [NetInf](#) – is depicted in Section 3.8.

3.1 Data Model

This section provides a detailed overview of the architecture and implementation of the data model used in [NetInf](#).

3.1.1 Architecture

Our central approach is to organize information in so-called InformationObjects ([IOs](#)). Those [IOs](#) hold both the actual information and the metadata to manage the [IOs](#).

Types An [IO](#) is mainly a container for several attributes. Some attributes are mandatory, some are optional. Since our main representation for an [IO](#) uses Resource Description Framework ([RDF](#)) [9], the meaning of attributes is predefined with the help of a RDF Schema ([RDFS](#)) [8]. Which attributes are mandatory and which are optional depends on the type of the [IO](#). In our architecture there are three different types: an IdentityObject ([IdO](#)), a DataObject ([DO](#)), and a general [IO](#):

IdOs contain all information that is necessary to identify a unique person (or program) which generates information for [NetInf](#). Besides general [IdOs](#), there exist special [IdOs](#) which represent [NetInf](#) components: ResolutionServiceIdentityObjects represent [RSSs](#), SearchServiceIdentityObjects represent [SSs](#). These special [IdOs](#) are,

e.g., used to describe the single services, to store configuration data or to allow a service to subscribe to events at an [ES](#).

DOs are special [IOs](#) which refer to external data/information. This data can, for example, be a web resource. See Section 5.1 for details on how **DOs** and the contained `http://url` attributes are used.

IOs are the general type, which can contain any information.

Beyond those three kinds of [IOs](#), we talk about BitlevelObjects (**BOs**). **BOs** are raw data, e.g. referenced by a [DO](#)'s `http://url`.

Attributes An attribute consists of three parts: attribute identification (URL to the defining [RDFS](#) ontology), attribute purpose (SYSTEM, USER, LOCATOR), and the attribute value. The attribute identification is an URL to the corresponding [RDFS](#). Defining a comprehensive [RDFS](#) for [IOs](#) and attributes is still to be done, see Section 8.1.1.1 in the *Open Issues* chapter. The attribute purpose depends on how the attribute should be used: USER attributes are for general use, SYSTEM attributes for security and integrity evaluation, and LOCATOR attributes for referencing other resources within [NetInf](#) or in the Internet. The attribute value consists of the data type (e.g. String, Integer, Boolean, ...) and the actual value separated by a colon. Attributes may be nested. The inner attributes are called subattributes. Listing 3.1 shows an example attribute in RDF/XML notation.

Listing 3.1: Example `http://url` attribute (LOCATOR) pointing to the W3C website

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:netinf="http://rdf.netinf.org/2009/netinf-rdf/1.0/#">

    <netinf:http_url rdf:parseType="Resource">
        <netinf:attributePurpose>LOCATOR_ATTRIBUTE
        </netinf:attributePurpose>
        <netinf:attributeValue>String:<a href="http://www.w3c.org">http://www.w3c.org</a>
        </netinf:attributeValue>
    </netinf:http_url>

</rdf:RDF>
```

Identifier Each [IO](#) has a unique reference, its identifier. The identifier consists of a sequence of so-called identifier labels, which are separated by a tilde. The following identifier refers to an [IO](#):

```
HASH_OF_PK=8c4e559d464e38c68ac6a9760f4aad371470ccf9
~HASH_OF_PK_IDENT=SHA1~VERSION_KIND=UNVERSIONED
```

The label `HASH_OF_PK` contains the hashed public key of the `IO`'s creator. The label `HASH_OF_PK_IDENT` contains the name of the hash-algorithm that is used to hash the public key. Finally, `VERSION_KIND` defines whether the `IO` is versioned or unversioned. The user identified by this `IdO` can now create a new `IO` and assign a unique name, e.g. `MyToDoList`, to it. The identifier of this new `IO` would be exactly the same as the identifier referring to his `IdO`, but followed by `UNIQUE_LABEL=MyToDoList`.

For all `IO` types, the label `HASH_OF_PK` and `HASH_OF_PK_IDENT`, which are the hash of the owner's public key respectively the name of the hash-algorithm used for this hash, are mandatory. This is used for the integrity check, see Section 3.7.1. Furthermore, all types must have a `VERSION_KIND` label, whose value is either `VERSIONED` for versioned or `UNVERSIONED` for unversioned `IOs`. This is for future use, see Section 8.2.3. `IOs` and `DOs` must have an `UNIQUE_LABEL` label, which is an unique name to identify the actual information. All other labels in the identifier are optional. You can use any alphanumeric String (including underscores) for new labels, you may require in your software.

3.1.2 Implementation

The idea behind our data model implementation was interchangeability and the possibility of quick development in many places at once. As we wanted to get to work on the components actually using the data model as fast as possible, we decided to first specify interfaces for all data model objects and to write a basic implementation (called *impl* or *Java implementation*). We introduced a wrapper interface for all data model related objects, the `NetInfoObjectWrapper`, which has to be implemented by all data model classes representing transferable objects, regardless of which implementation. The wrapper interface provides basic methods for serialization such that communication components could easily be developed using only this interface. After some time, the `RDF` implementation became our main focus and is mainly used in most parts of `NetInf`. Another reason for the native Java implementation was the fact that Google Android does not support the `RDF` library *Jena*¹, which we are using. That meant that the `RDF` implementation was unavailable on mobile devices.

3.1.2.1 Package Structure

The implementation of the `NetInf` data model can be found in the package `netinf.common.datamodel`. This package is subdivided as follows

netinf.common.datamodel The main package containing the data model interfaces (`Identifier`, `IdentifierLabel`, `InformationObject`, `DataObject`) as well as factories and enums for predefined attribute purposes, label names and version kinds. The `NetInfoObjectWrapper` is also located here.

¹<http://openjena.org/>

netinf.common.datamodel.attribute Contains the `Attribute` interface and an enum for predefined attribute identifications.

netinf.common.datamodel.creator Contains the `ValidCreator`, a convenience class with which one can easily create valid data model objects. This is required as security and integrity functionality need certain attributes to be present but our generic data model approach does not allow for them to be present. There are more restrictions on how a valid data model object has to look like. They have been taken into account here.

netinf.common.datamodel.identity Contains the `IdentityObject` interface as well as special types of `IdOs` for [NetInf](#) components.

netinf.common.datamodel.impl.* Contains a concrete implementation of the data model based on Java serialization (referred to as *impl* or *the Java implementation*) as well as a *Guice* module (see Section 2.4) for it.

netinf.common.datamodel.rdf.* Contains a concrete implementation of the data model based on a [RDF](#) representation (referred to as *the RDF implementation*) as well as a *Guice* module for it.

netinf.common.datamodel.translation.* Contains the `DatamodelTranslator` which can translate between [RDF](#) and Java implementation as well as a *Guice* module for this functionality.

For further, detailed information about the data model implementation(s) please refer to the Javadoc comments located in the source files of the package or available as HTML. Those comments give you a detailed description of how the data model classes work together.

3.2 Node Interface

The described components of our [NetInf](#) architecture share a common foundation that provides functionality needed by [NetInf](#) Nodes, [ESs](#) and [NetInf](#) Applications. All this functionality resides in a *Commons Library* which essentially consists of the following parts:

- The data model
- The security framework
- Communication functionality

The data model mainly describes the structure of information that can be stored in **NetInf** and has already been described in Section 3.1. The security framework provides features such as integrity verification and owner authentication of **IOs** and is described in detail in Section 3.7.

Of particular interest is the communication functionality that provides means for message exchange between **NetInf** Nodes and between **NetInf** Nodes and an **ES**. Hence, it defines our NODe API and is the topic of this section.

3.2.1 Architecture

NetInf supports the TCP/IP transport protocol and multiple encodings for the message contents. This enables the communication between **NetInf** Nodes or with an **ES** and allows the exchange of more than one request/response pair.

The supported message encodings are Google Protocol Buffers² – in short *protobuf* – on the one hand and XML on the other hand. Protobuf provides a simple and highly-performant binary encoding of structured data. But since the encoded messages are not human-readable and because a sophisticated protobuf library does not yet exist for all major programming languages, **NetInf** can also handle messages encoded as XML. The resulting XML message is typically more bloated than the protobuf equivalent and parsing it requires more computing power. Hence, protobuf should be used where possible.

Note The node provides technically a HTTP interface as well. This interface was previously developed to enable the communication with an application like InFox, but has been replaced by the Application Interface (see Section 3.3).

3.2.2 Implementation

What has been described so far is implemented in the Java package `netinf.common`, or to be more precise, in the two packages `netinf.common.communication` and `netinf.common.messages`. The former contains functionality to send and receive messages and the latter contains the message classes themselves.

All messages inherit from the abstract class `NetInfMessage`, for example the `RSGetRequest` that fetches an **IO** from a **RS**. For each request message there is an according response message, in the mentioned example this is the `RSGetResponse`. These are the message classes that **NetInf** Nodes, **ESs** and **NetInf** Applications should use. They will be automatically encoded as protobuf or XML when they leave the local **NetInf** Node, **ES** or **NetInf** Application.

The package `netinf.common.communication` implements this automatic encoding and decoding as well as basic functionality to send and receive **NetInf** Messages synchronously and asynchronously via the `Communicator` class. An encoded **NetInf** Message is prefixed with the ID of the encoder that has been used such that the recipient knows which decoder it has to use – such a tuple (encoder id, encoded

²<http://code.google.com/p/protobuf/>

message) is represented by the class `AtomicMessage`. The `MessageEncoder` interface is currently fully implemented by `MessageEncoderProtobuf` and partially by `MessageEncoderXML`. The tag names of the XML message format are class constants of `MessageEncoderXML` but have not yet been formalized as a DTD or XSD.

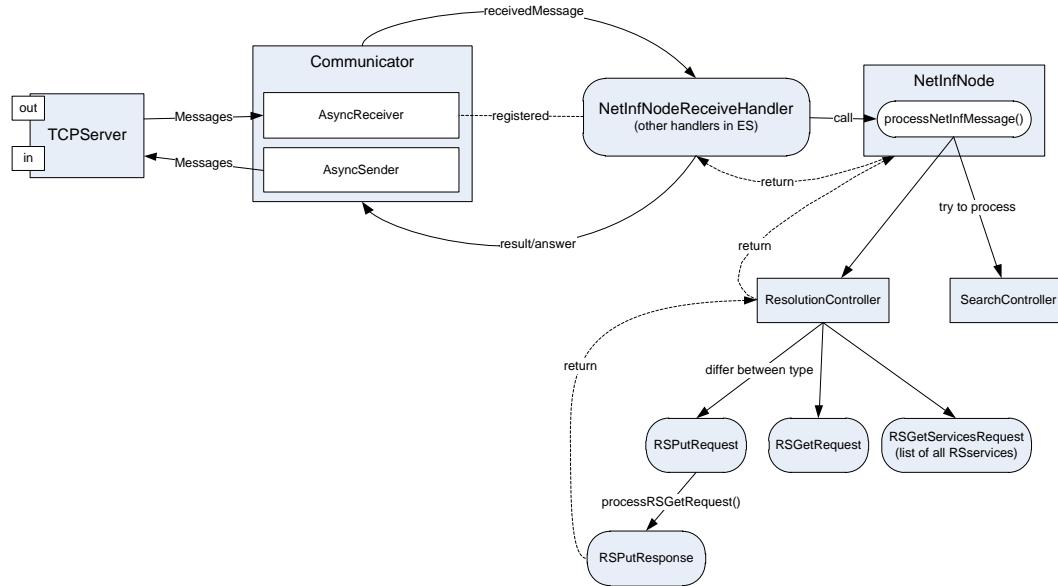


Figure 3.1: Overview of the internal Node Communication

Figure 3.1 shows how the communication within a `NetInf` node works internally. `NetInf` messages are received via the `TCPServer` which is observed by an `AsyncReceiver`. This receiver is held by the already mentioned `Communicator` that forwards the message to the so-called `NetInfNodeReceiveHandler`. Within a `NetInf` node the attached controllers and related services try to process the received message regarding the particular type. If everything works fine the response of that message is created and returned the whole way back. The `AsyncSender` of the `Communicator` sends the response message via the `TCPServer` back.

3.2.3 Protobuf Messages

A detailed description of the technical protocol used for the communication can be found within the code, and the Javadoc derived from the code. Notice that the protobuf messages which are specified in the file `ProtobufMessages.proto` in the package `netinf.common.communication.protobuf` do not have any documentation attached to them. But each protobuf message has an accordingly named Java class in the package `netinf.common.messages` with identical parameters and Javadoc.

3.2.4 Convenience Features

Since it is a very common task to e.g. fetch an **IO** or to issue a search request, some of these very typical interaction patterns have been abstracted and simplified further. The class `RemoteNodeConnection` has such convenient methods that furthermore automatically perform security-related checks like integrity verification. In addition to that, it automatically re-establishes a closed connection.

getIO Fetches the **IO** with the given identifier

getIOS Fetches all **IOs** with the given identifier (i.e. all versions)

putIO Creates a new **IO** or modifies an existing **IO**

deleteIO Deletes an **IO**

performSearch Returns the identifiers of those **IOs** that match a SPARQL query

3.3 Application Interface

In order to simplify the access to a **NetInf** Node and to make it independent of the Java package `net.inf.common` and the respective **NetInf** messages an additional interface, the application interface, was introduced as a **NetInf** component.

The application interface is intended for application developers who are neither able nor willing to implement **NetInf**'s node interface. Since the communication is based on HTTP, a common Web standard, even already existing applications can harness the capabilities of an information-centric network.

In the following the application interface and its usage are described.

3.3.1 RESTful HTTP

The application interface implements a RESTful API following the architectural style called Representational State Transfer (REST) [6].

Accordingly, three different resources were identified in **NetInf** and made accessible through an HTTP interface. In the following these resources are specified in detail. This includes how individual resources are addressed and what operations, i.e., HTTP methods / verbs, are permitted. All method definitions are listed in the HTTP/1.1 specification (see RFC 2616³).

IO Resources

The first resource type represents **IOs**, and therefore, **IdOs** as well as **DOs** in their respective RDF formats. IO resources are identified by the following URI pattern:

³<http://tools.ietf.org/html/rfc2616>

3 Components in Detail

```
/io/ni:HASH_OF_PK={hashOfPK}~HASH_OF_PK_IDENT={hashOfPKIdent}~  
VERSION_KIND={versionKind}~UNIQUE_LABEL={uniqueLabel}~  
VERSION_NUMBER={versionNumber}
```

Requests pointing to such a path will be redirected according to the following URI pattern:

```
/io?hash_of_pk={hashOfPK}&hash_of_pk_ident={hashOfPKIdent}&  
version_kind={versionKind}&unique_label={uniqueLabel}&  
version_number={versionNumber}
```

IO resources permit the use of all five HTTP methods, i.e., GET, HEAD, POST, PUT, and DELETE. This set of operations may be used as a replacement for the management tool.

BO Resources

A **BO** resource represents the actual data that a **DO** can refer to. Instead of just handing back one possible locator to a **BO** the application interface takes care of downloading and returning the actual data. BO resources are identified by the following URI pattern:

```
/ni:HASH_OF_PK={hashOfPK}~HASH_OF_PK_IDENT={hashOfPKIdent}~  
VERSION_KIND={versionKind}~UNIQUE_LABEL={uniqueLabel}~  
VERSION_NUMBER={versionNumber}
```

Notice the absence of a /bo/ prefix. Since the main purpose of the application interface is to provide the actual data a shorter path was chosen. But such requests will be redirected according to the following URI pattern (cf. IO resources):

```
/bo?hash_of_pk={hashOfPK}&hash_of_pk_ident={hashOfPKIdent}&  
version_kind={versionKind}&unique_label={uniqueLabel}&  
version_number={versionNumber}
```

Currently **BOs** can only be retrieved via the application interface, i.e., only GET and HEAD are supported.

Search Resources

A search resource offers a search mask which simple queries to **NetInf** can be sent with. Search resources are identified by the following URI pattern:

```
/search?query={query}
```

Instead of providing a SPARQL query the search term is interpreted as a regular expression and used in a predefined SPARQL query applying the regular expression to all attributes of an **IO**.

Search results are presented as a list of **NetInf** identifiers referencing to the respective **IO** and **BO** resources.

3.3.2 Legacy Support

As mentioned earlier even already existing applications that are not capable of sending **NetInf** messages directly can make use of **NetInf**. In the context of **NetInf**, such applications are called legacy applications. In the following an example for the use of **NetInf** with such a legacy application is presented.

Video Streaming with VLC

The VLC media player⁴ does not natively support communicating with a **NetInf** node. But because VLC supports HTTP it is possible to stream a video that is only accessible as **NetInf DO** via the application interface.

Figure 3.2 depicts the steps that are necessarily executed to play a video in VLC or in a browser capable of video playback.

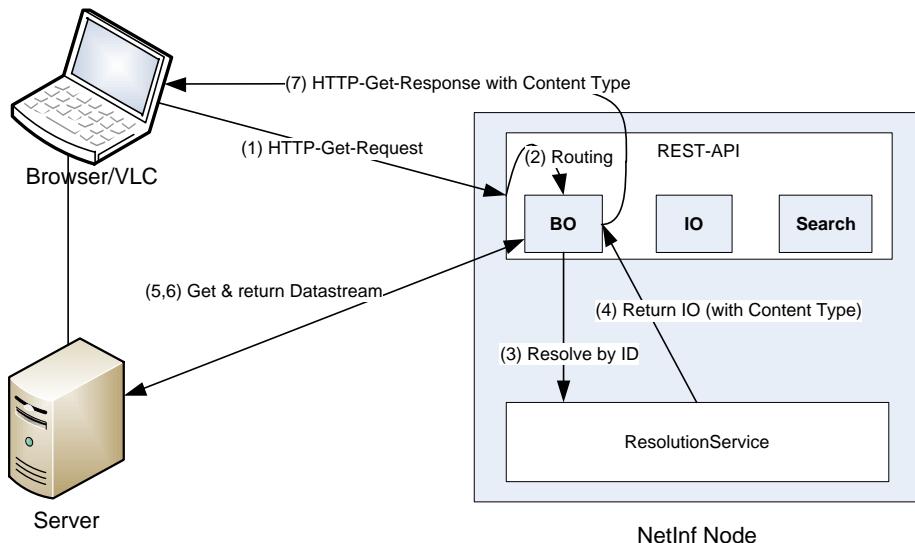


Figure 3.2: REST Functionality

3.4 Resolution

An essential part of **NetInf** is the resolution of identifiers to the actual information. As mentioned in Chapter 2, this is done by the Resolution Controller (**RC**) with the help of several Resolution Services (**RSS**s). Beside these two constituents, the resolution is supported by a **ResolutionServiceSelector** and a set of **ResolutionInterceptors**. In the following subsections, these components are described in more detail. An overview can be seen in Figure 3.3.

⁴<http://www.videolan.org/vlc>

3 Components in Detail

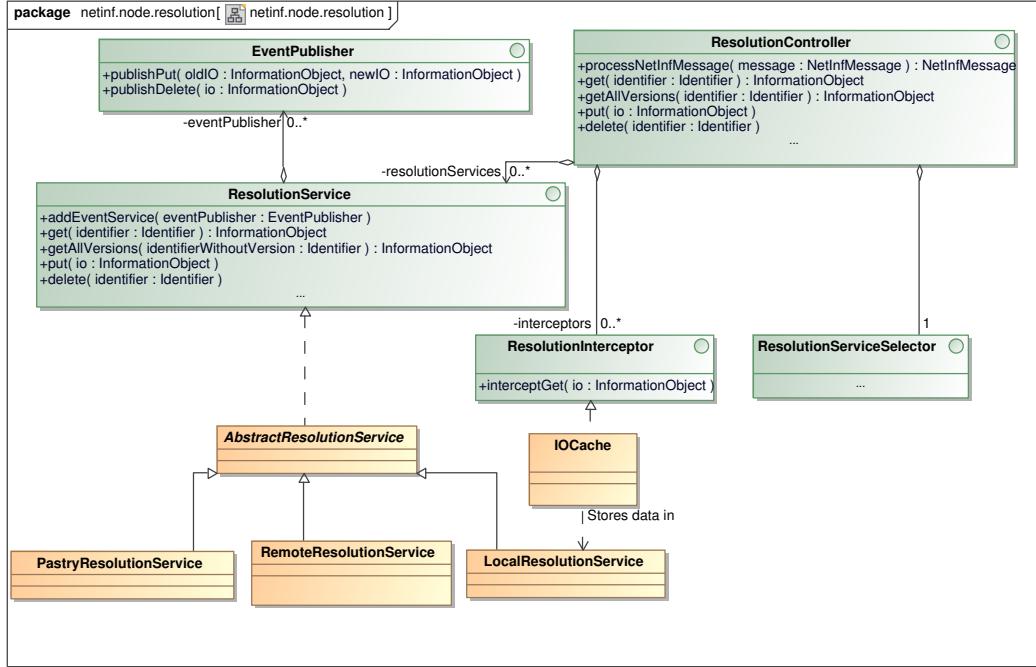


Figure 3.3: Resolution overview

3.4.1 Resolution Controller

The **RC** is the central part of the Resolution component. The **RC** manages a set of **RSs**. The actual set of **RSs** can be configured for each node with the help of *Guice* (see Section 2.4). The **RC** now gets messages from the communication part of the node (see Section 3.2.2). First, security and integrity of the messages are checked as described in Section 3.7. In the following, the defined **ResolutionInterceptors** are called. Then the messages are dispatched to method calls on **RSs**. The selection of the **RSs** used for a particular request and the order in which they are called is determined by the **ResolutionServiceSelector**. The results gained from the **RSs** are then wrapped in a response message and returned to the communications part.

3.4.2 Resolution Service

The **RSs** handle the actual put, get and delete operations of **IOs** in **NetInf**. The get operation takes an identifier and returns the actual **IO** for it. The put operation takes an **IO** and stores it in **NetInf**, and the delete operation removes the **IO** from **NetInf**. Furthermore, the **RSs** publish events to the **ES** when somebody creates, changes or deletes an **IO**. This is done with the help of the **EventPublisher**. A **RS** is represented by a **ResolutionServiceIdentityObject**, as mentioned in Section 3.1. Besides a description of the **RS**, the **IdO** stores a default priority for the **RS** which can be used by the **ResolutionServiceSelector**.

In our prototype there are several **RSs** implemented. A local **RS** which can be used to store **IOs** on a local computer (**LocalResolutionService**), a dis-

tributed **RS** which uses an underlying *Pastry*⁵ ring for resolution and storage of the **IOs** (*PastryResolutionService*) and a remote resolution service which can be used to forward the request to another **NetInf** Node (*RemoteResolutionService*). Furthermore an **MDHT** system has been introduced, which allows distributed resolution of **IOs** globally on multiple levels. The bootstrapping of this *RemoteResolutionService* can either be done using a configuration file with the address and port of a **NetInf** Node or using **GP** as described in Section 4.2. Furthermore, we developed a **RDFResolutionService** which uses RDF as the underlying data format and is therefore especially useful as a data store for the Search component (see Section 3.5.4). To simplify the development of new **RSs** we provide the **AbstractResolutionService** which can be used as a starting point.

3.4.2.1 MDHT

The *Multilevel Distributed Hash Table (MDHT)* is designed to be a global name resolution system for **NetInf** nodes. It allows a client to send requests for specific Information Objects (**IOs**) - of which **DOs** are a subclass - and receive a response with the complete **IO**, if it exists, or an empty response if it does not exist in the system. The main idea (see original paper [4]) is to keep the resolution requests local, so that the most resolution requests are resolved locally. First an overview of the system will be given, followed by developer considerations and needed development work.

Overview The **MDHT** System is composed of multiple *Distributed Hash Table* sub-components - at least one - running at different layers. Layers and Distributed Hash Tables (**DHTs**) are 1:1 – only one **DHT** (e. g. FreePastry) can be a layer at any time. From this moment on the terms “Layer” or “Level” and “**DHT**” will be used interchangeably. As a convention, the **MDHT** layers are numbered incrementally, from 1 to n (natural number), with 1 being the “lowest” - as in the nearest to the client - and n being the “highest” - as in furthest from the client.

At least one node is necessary to start a **DHT** (seed node). An **MDHT** node will start a seed node by itself as the default behavior. This can be changed by specifying another node that the node should join to. It is not absolutely necessary that the **DHTs** are exactly the same type (e.g. *FreePastry* or *Chord*), see the developer notes below for how to develop your own **DHT** component for **MDHT**. They must, however, adhere to the CommonAPI⁶ standards for P2P APIs.

An **MDHT** node may be part of several different layers at the same time. To better illustrate its behavior a description of the resolution process is given here (all nodes are **MDHT** nodes, unless otherwise specified):

1. Node A [Level 1] receives a “get” request with an **IdO**.
2. Node A [Level 1] searches in its local **DHT** storage (e.g. PAST storage) to see if it has the **IO** corresponding to the given **IdO**. Note that the **IO** does not have

⁵<http://www.freepastry.org/>

⁶<http://pdos.csail.mit.edu/papers/iptps:apis/main.ps>

to be stored on the current node, the **DHT** just retrieves the stored object from the corresponding node (e.g. assuming the IdO is nearest in ID space to Node B [Level 1], then the PAST storage component running on top of the **DHT** will retrieve the object from Node B [Level 1] in a way that is completely transparent to the **MDHT** system) If it is present the node returns the **IO** to the requester and stops the resolution process.

3. If the **IO** is not present in the local storage of Node A and there are more than 1 **MDHT** levels: Node A [Level 1] switches to the next level. Node A [Level 2] now searches the local PAST storage (i.e. all the nodes on this level) for the **IO**. If it is present, it is then returned to Node A [Level 1] along with an acknowledgment (ACK message), which in turn returns it to the client.
4. If the **IO** is not present on this level either, the search is taken to the next level in the same manner above, until either the level threshold has been reached (if it was present in the “get” request) or until the maximum **MDHT** level has been reached. Should the **IO** not be found at any level, the response will be a null object.

Assumptions Some assumptions you should be aware of:

- All nodes are present at all levels i.e. there is no node at Level 1 which is not reachable at the maximum Level. This makes the system global and avoids isolated partitions.
- The node sending the response to the requester is always the **MDHT** node nearest to it.
- Resolution requests are at the time **BLOCKING**. This means that one has to finish before the other can start, in case there are more of them. This should probably be changed.
- The ACK message is currently a `NetInfMessage`. This should probably be changed.

Developer Notes The class diagram for the **MDHT** system can be seen in Figure 3.4
The **MDHT** Resolution Service implementation can be found in the `MDHTResolutionService.java` file. To create a Google Guice module with the **MDHT** simply insert the following code in your java file:

In the `configure()` method:

```
// other install directives omitted here
// ResolutionServices
install(new MDHTResolutionModule());

// ...
// other providers omitted here
```

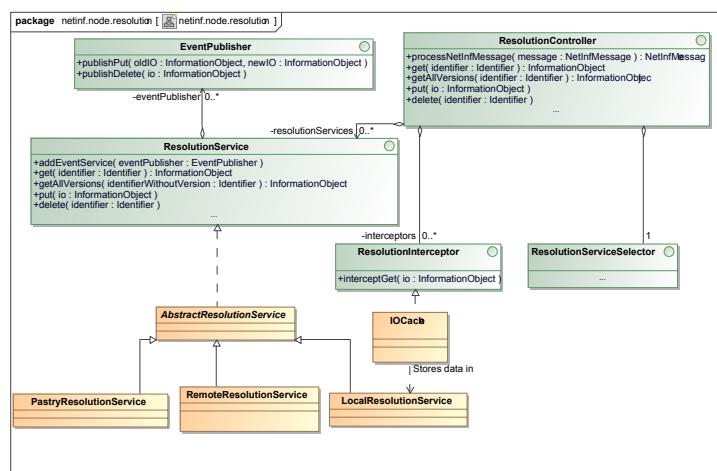


Figure 3.4: MDHT Classes

```
@Singleton
@Provides
ResolutionService[] provideResolutionServices(
    RemoteResolutionFactory remoteResolutionFactory,
    MDHTResolutionService mdhtResolutionService) {
    ResolutionService[] otherRS = { mdhtResolutionService
        };
    return otherRS;
}
```

An example of a module can be found in `MDHTRSPsCacheNodeModule.java`.

You can then launch the node by using the `StarterNode` as the main class and the Guice Module as an argument. The same thing can be achieved in Eclipse by creating a launcher configuration. Otherwise a command line run will also do (assuming you've already used Ant to create a `node.jar` file):

```
java jar [path-to-node]/node.jar netinf.path.to.my.module.
    ModuleName
```

Cautionary Notes The acknowledgment (or ACK) message sent by the storing **MDHT** node to the node where the “put” request originated is a custom NetInf message implemented in the class `netinf.common.messages.RSMDHTAck`. This message is XML encoded and serialized in the RDF format. For the prototype this is enough, but in order for the messaging scheme to be consistent in the future, we recommend reviewing how and what messages are being sent. An example of inconsistency: The ACK message does not require a response, but all other NetInf messages do. Initially, all NetInf messages were thought as response/reply, but a redesign needs to take place to consistently accomodate such (and future) messages. Also note that this creates a dependency on **NetInf**: an **MDHT** node needs to also be a **NetInf** node.

Another issue is the **DHT** interface in `netinf.node.resolution.mdht.dht.DHT`. While it is general it still needs to be checked for CommonAPI⁷ compliance. Any future **DHT** implementation implementing this interface should be **MDHT**-compatible, so it is important that the interface itself is consistent and does not depend on some proprietary P2P library. This might not be so easy anyway: According to our research, the number and quality of existing Java-compatible P2P APIs is sadly very low. An important design decision should be taken here to accomodate other **DHTs** in the future.

3.4.2.2 RDF

The RDF Resolution Service stores the InformationObjects in their **RDF** representation in a local SDB database – based on Jena.

If the `SearchServiceRDF` is configured to use the same database, it can be used to search for **IOs** within this resolution service. This resolution service does not publish

⁷<http://citeseerrx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.5548&rep=rep1&type=pdf>

`put()` or `delete()` operations to an event service. The **RDF RS** supports working with versioned **IOs**. It currently supports searching the **IO** database by using SPARQL queries.

All the options necessary to initialize the RDF service in the module properties file start with “`rdf.`”, for instance the database name is represented by the option “`rdf.db.sdbName`”. This is configurable in the `SDBStoreFactory.java` file.

The only inconvenience with the **RDF RS** is that it cannot globally resolve queries (i.e. if there are two nodes, A and B, and both have the **RDF RS** active, the **IOs** put by A cannot be seen by B, unless they are somehow using the same database). This creates, however, a bottleneck for the system and has to be architecturally reconsidered. For the prototype this is however sufficient.

3.4.2.3 Pastry

For our **DHT** implementation we have used *FreePastry*, a Java implementation of which is available online, but has not been developed since 2009. For use with the **MDHT** we have customized the *PAST* layer and added a special *NetInfPAST* application to each Pastry node. The implementation is in `NetInfPast.java`. In FreePastry, each node has its basic routing infrastructure (i.e. **DHT**-based) support. On top of this layer other so-called *applications* are running, which need to be registered on the node. The registration takes place e.g. in the constructor of our `FreePastryDHT` class as follows:

```
past = new NetInfPast(pastryNode, storageManager, 2, pastName,
                      this);
```

The above lines of code create a new `NetInfPast` application with a given `StorageManager` (used to specify how to generate the IDs of stored objects, what type of storage to use and what type of caching to use). See the `FreePastry` source code for `rice.persistence.StorageManagerImpl` for more documentation and details. Sadly the entire `FreePastry` library is poorly documented, but the source code is somewhat understandable if you are willing to spend the extra hours. You may also choose to use `FreePastry` wrappers such as `Bunshin`⁸ in the future.

This is how the `FreePastry DHT` interacts with the **MDHT** in the `put` request (Figure 3.5) and in the `get` request (Figure 3.6).

We have also customized the *Insert* and *Lookup* messages for Pastry, as we needed them to support the **MDHT** (added a new `Level` attribute). The respective messages can be found in the classes `NetInfInsertMessage` and `NetInfLookupMessage` respectively. The **IOs** are now stored as custom *PAST* objects, the definition of which can be found in the class `MDHTPastContent`.

If you wish to extend the existing **DHT** make sure to spend some time with the code and the Javadoc within, as well as with the above diagrams. Also try to turn on debug logging on *multiple* nodes, not just one (3 should be enough). For `FreePastry` to log debug messages you need to enable it like this (taken from `FreePastryDHT`):

```
environment.getParameters().setString("logging_enable", "true")
;
```

⁸<http://planet.urv.es/bunshin/>

3 Components in Detail

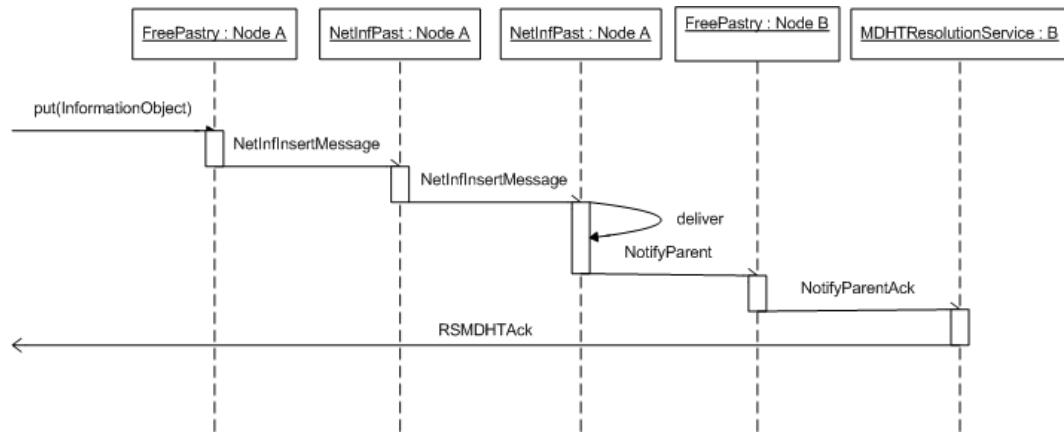


Figure 3.5: MDHT Put Sequence

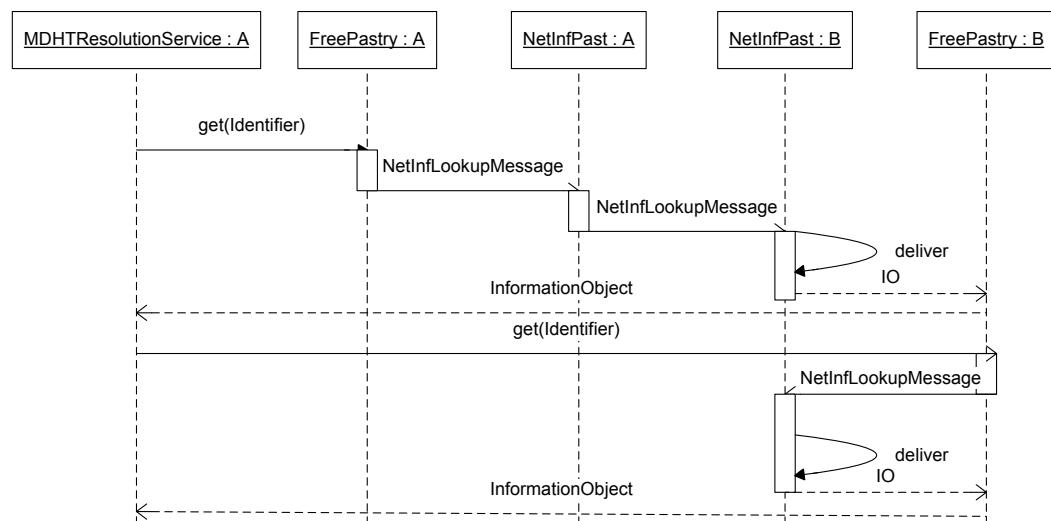


Figure 3.6: MDHT Get Sequence

```
environment.getParameters().setString("loglevel", "DEBUG");
```

3.4.3 Resolution Service Selector

In order to externalize the decision which **RS** is used to handle a specific request, we provide a `ResolutionServiceSelector` interface. This interface has to provide an ordered list of **RSs** for put, get and delete requests. The `ResolutionServiceSelector` can easily be exchanged with *Guice* modules.

3.4.4 Resolution Interceptor

In order to allow the adaption of the resolution process to specific needs, we provide the possibility to add `ResolutionInterceptors` to the resolution process. These interceptors will be called before the request is dispatched to the **RSs**. In our prototype, the `ResolutionInterceptors` are mainly used to plug in caching functionality and do some special handling for **DOs**. The handling of **DOs** and binary content is described in Section 3.6.

Caching **IOs** from the distributed **RS** or from other nodes can be cached for faster retrieval or for situations where these services cannot be reached. Caching of versioned **IOs** can be done unproblematically because versioned **IO** are immutable such that the cached content will never be invalid. For mutable content the **ES** could be used for cache invalidation. The caching solution implemented in the previous prototype was based on a `LocalResolutionService` which was populated with **IOs** obtained from or pushed into other **RSs**. The current prototype caches **IOs** inside the a custom implementation of the PAST distributed storage layer in the **MDHT** system and does not use other services. The cached objects are only session-persistent, which means if the `MDHTResolutionService` is restarted, the cached **IOs** are lost, but during the session they are persisted in RAM. There is no caching strategy in place, currently all **IOs** are cached. For use in production more sophisticated caching strategies should be used and can be implemented for instance in the custom PAST storage layer – in our prototype this is called `NetInfPast`.

3.5 Search

This section describes the Search component of **NetInf**. As already mentioned in Chapter 2, the search feature is one of the three services a **NetInf** Node can offer. Search in this case means that one can search for specific **IOs** with respect to their attributes. The result of a search request is a list of the identifiers of the **IOs** that matched the search query. After getting the search results, the caller may use the Resolution component (see Section 3.4) to fetch the **IOs** that have been found.

The following subsections show how the Search component is interrelated with other components of **NetInf**, of which classes the Search component consists and how it op-

erates. Finally, some information about the `SearchServiceRDF`, which is our prototype implementation of a Search Service ([SS](#)), is given.

3.5.1 Overview

This subsection gives an overview of the Search component which shows its basic components and their interrelation to other parts of the `NetInf` architecture (see also Figure 3.7).

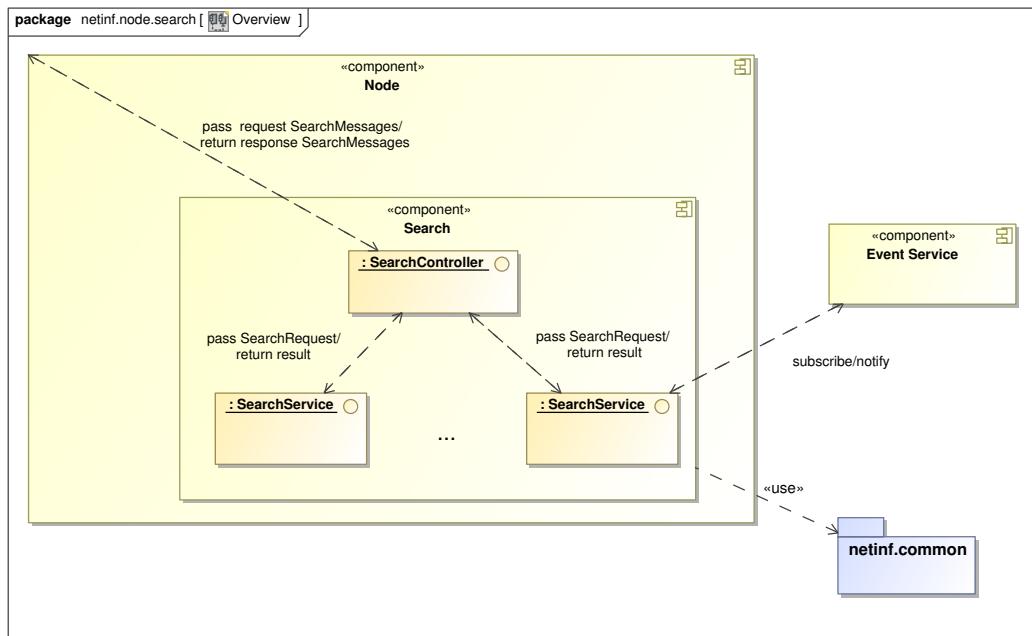


Figure 3.7: Overview of the Search component with its basic components and their interrelation to other parts of the architecture

As already mentioned, the Search component is an optional part of a `NetInf` Node. It mainly consists of a `SearchController` and an arbitrary amount of `SearchServices`. Incoming search requests are passed from the node to the Search Controller ([SC](#)). The controller handles the request and forwards it to all available [SSs](#).

There are several possibilities for a [SS](#) to build up the data basis for its search. For example, a [SS](#) may be able to search within the storage of a single [RS](#). Another elegant solution is the following: [SSs](#) may be connected to an [ES](#) (see Section 3.8). If they are, the [ES](#) notifies them of (changes of) the [IOs](#) that are stored in [RSs](#) which publish to the [ES](#). In this way, [SSs](#) can act as global [SSs](#) always informed of the current content of [IOs](#). Depending on whether they subscribe to all [IOs](#) or only [IOs](#) with specific characteristics, they act as a general or a specialized [SS](#).

As all parts of our `NetInf` solution, the Search component uses parts of the `netinf.common` library (see Section 3.2).

3.5.2 Class View

Figure 3.8 shows the internal architecture of the Search component in more detail. One can see the mainly involved interfaces and classes and their main methods (without parameters).

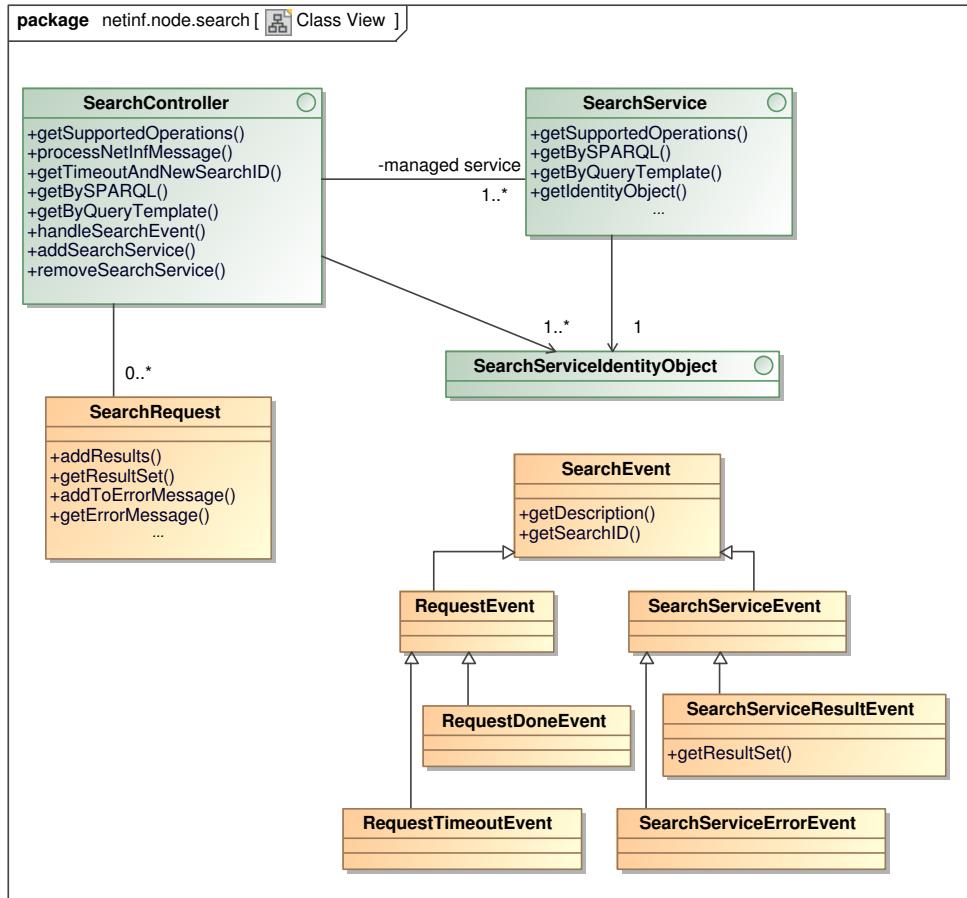


Figure 3.8: Class view of the Search component

An implementation of a **SC** has to implement the `SearchController` interface. Similarly, each **SS** implementation has to implement the `SearchService` interface. A **SC** manages at least one **SS**.

SSs allow two types of query specification. On the one hand, search queries can be written in SPARQL [11], a query language for **RDF** data which is comparable to SQL. An example for a query which selects all **IOs** that have an attribute with the attribute identification *name* and the attribute value *foobar* is given in Listing 3.2.

It is only necessary to specify the WHERE part of the query. The variable `?id` already points to the identifier within the **RDF** representation of **IOs**. Regarding the example this means that only lines 4 and 5 have to be specified. This helps to make the query specification a bit more user friendly. Furthermore, it assures that the results are always identifiers. To conclude, using SPARQL is a flexible and powerful possibility to formulate search queries.

3 Components in Detail

Listing 3.2: Sample query for the search query specification using SPARQL

```
PREFIX netinf: <http://rdf.netinf.org/2009/netinf-rdf/1.0/#>
SELECT ?id WHERE {
?blankNode netinf:transportedIO ?id.
?id netinf:name ?blankNode2.
?blankNode2 netinf:attributeValue 'String:foobar'.
}
```

On the other hand, **SSs** can have built-in query templates for search requests such that the caller only has to specify the actual values for some parameters. For example, Listing 3.3 shows the SPARQL query to search for all **IOs** with a specific owner.

Listing 3.3: SPARQL query for searching for all IOs with a specific owner

```
PREFIX netinf: <http://rdf.netinf.org/2009/netinf-rdf/1.0/#>
SELECT ?id WHERE {
?blankNode netinf:transportedIO ?id.
?id netinf:owner ?blankNode2.
?blankNode2 netinf:attributeValue '<identifierOfOwner'sIdO>'.
}
```

To simplify such a search request, one could implement a query template which takes the identifier of the owner's **IdO** as a parameter. Now, the caller only has to specify the name of the template and the specific identifier of the owner. The SPARQL query is hidden within the **SS**.

Of course, the usefulness of implementing and using a query template increases if the SPARQL query gets more complicated. Generally, all query templates and their required parameters should be listed within the `DefinedQueryTemplates` enum, which is located in the `netinf.common.search` package. Nevertheless, a specific **SS** may not support all templates defined in this enumeration.

A **SS** is represented by a `SearchServiceIdentityObject`, as mentioned in Section 3.1. This representation is used for describing the **SSs**, for the management of the **SSs** by the **SC** (see Section 3.5.3), and to allow **SSs** to subscribe to events at an **ES**.

Each search request is processed in parallel by the different **SSs**. So each time the **SC** hands over a request to the services, it starts a new thread for each service to process the request. Thus, we need concepts on how to report results from the services to the controller and how to handle these results at the controller.

Search Events **SS**s inform the **SC** about their search results by handing over an instance of a subclass of `SearchServiceEvent`. A **SS** has to hand over exactly one of these “events” as soon as it finishes or aborts processing the request. It hands over a `SearchServiceResultEvent` object in case the processing was successful. The object contains the (possibly empty) result set. When an error occurred and the service is not able to report a result, it hands over a `SearchServiceErrorEvent` object. This object contains an error message and informs the controller that the service will not report any result for the search request.

Besides the subclasses of `SearchServiceEvent`, there exists another group of “events”, namely subclasses of `RequestEvent`. Instances of these classes inform the controller that a search request is finished and that it can return the results to the caller. Two such classes exist. The `RequestDoneEvent` class is used when all services reported a result (or an error). Furthermore, since the processing of a search request may take a long time or a service may not send a `SearchServiceEvent` object, each search request has a maximum processing time (timeout). The **SC** starts a timer for the timeout. In case this timeout is reached, the timer hands over a `RequestTimeoutEvent` object to the **SC**.

Search Request Representation To handle all the data that belongs to a search request, the **SC** uses the `SearchRequest` class. Each search request is represented by an instance of this class. Each search request is identified by a search ID, thus this class stores this ID. Among other things, the **SC** uses this class to store the results and error messages that it receives from the services. When all services finished processing or the timeout occurred, the controller fetches the results (or the error messages if at least one exists and the result set is empty) from the `SearchRequest` and returns them to the caller.

3.5.3 Operational View

This subsection describes how the **SS**s are managed by the **SC** and what the processing of a search request within the Search component in general looks like.

Search Service Management The **SC** has to somehow identify and administer its managed **SS**s. In addition, each **SS** is represented by a `SearchServiceIdentityObject`. When starting the node, all **SS**s which should run on the node are added to the **SC** by calling its `addSearchService(SearchService service)` method. Within this method the controller calls the `getIdentityObject()` method of the service to get its **IdO** and stores the received **IdO** along with the reference to the **SS** instance.

Later on it uses the **IdO** to differentiate between the **SS**s. For example, it stores the **IdOs** of all involved **SS**s and their search status within a `SearchRequest` object. Only if the controller wants to call a method of the service, it uses the concrete object instance.

Search Request Processing In the following, the processing steps of the Search component for an incoming search request are described. This description focuses on the main steps, so some methods and parameters that are not necessary for the main idea are omitted. In addition, Figure 3.9 illustrates the necessary processing in a sequence chart.

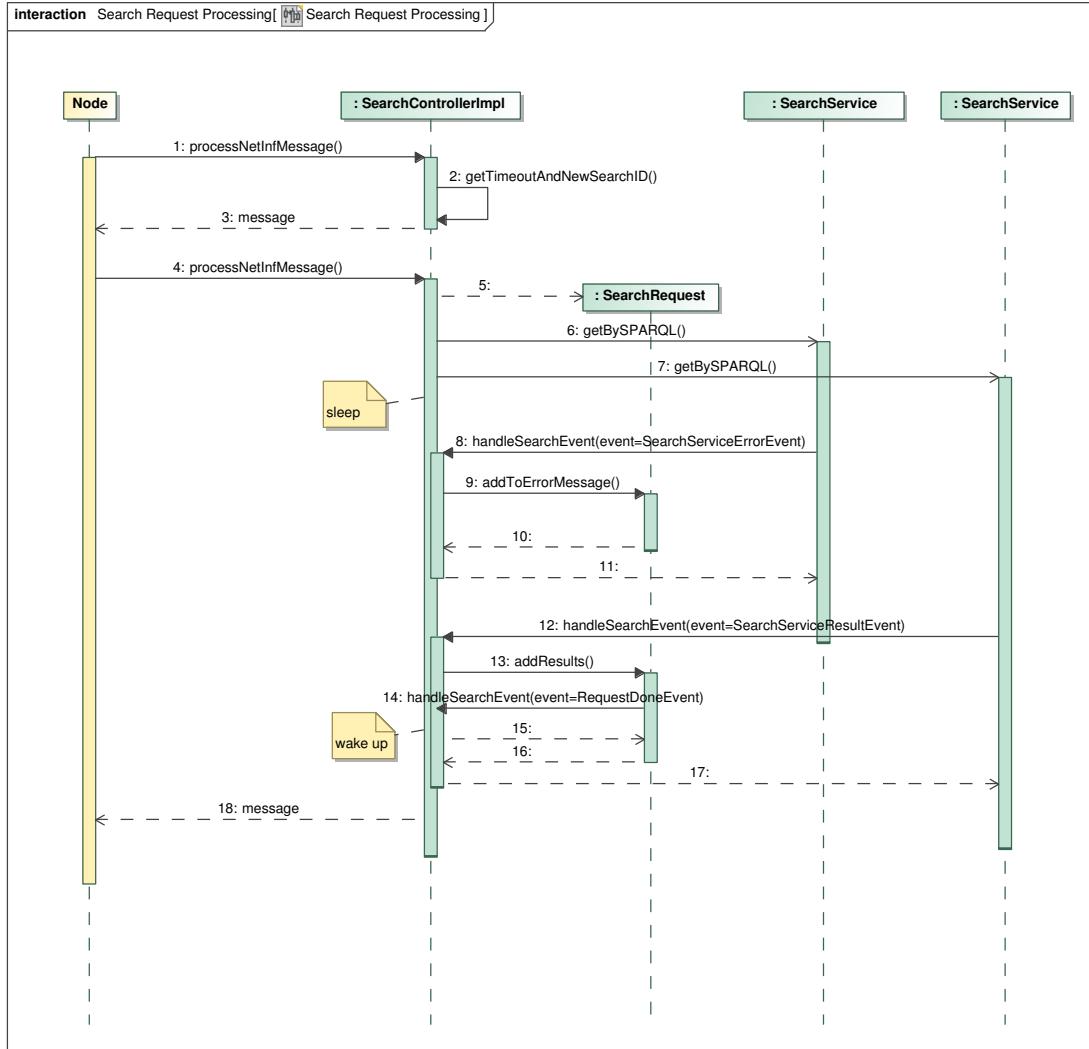


Figure 3.9: Processing of a search request within the Search component

For the initial situation it is assumed that the node including its Search component is running and that the **SC** manages two **SSs**.

A search request consists of sending two **NetInf** Messages. The first one is needed to initialize the search request, the second one to send the actual search query.

The request is initialized by sending a `SCGetTimeoutAndNewSearchID-Request` message with the timeout the caller desires. The node forwards it to the **SC** by calling `processNetInfMessage(...)` (step 1). The **SC** evaluates the actual timeout for this query by calculating the minimum of desired timeout and internal

maximum timeout. Furthermore, the **SC** assigns a search ID for this search request. It responds with a message that contains the search ID and the timeout that will be used.

Now the request is initialized and the actual search query can be sent (in this example, the specification via SPARQL is used). The node again forwards this message to the **SC** by calling `processNetInfMessage(...)` (step 4). The controller checks, whether the message contains a valid search ID. If so, it creates a new `SearchRequest` object first. Then it starts a timer with the beforehand determined timeout. Afterwards, it forwards the request to all (in this case the two) **SSs** and starts their asynchronous processing (steps 6 and 7). The request is now processed in parallel by the services while the initial thread sleeps until it gets informed about the completion of the overall search request.

When a service finishes processing, it informs the controller about its result. Two cases are possible: The service searched successfully and maybe found some matching **IOs**, or an error occurred during the execution. The **SS** notifies the controller by calling its `handleSearchEvent(...)` method with the appropriate event type. Steps 8 to 11 show the error case: The method is called with a `SearchServiceErrorEvent` object which contains an error message. The controller adds this message to the appropriate `SearchRequest` object. In case of success (steps 12/13), the method is called with a `SearchServiceResultEvent` object which contains the (possibly empty) result set. In this case the controller adds the result set to the `SearchRequest`. In both cases, finally, the **SC** marks the **SS** as finished in the `SearchRequest`.

Each time results or an error are added to a `SearchRequest`, it is checked whether all services finished the processing of this request. If so, the `SearchRequest` calls the `handleSearchEvent(...)` method of the controller with a `RequestDoneEvent` object as its parameter. In this case, the method wakes the initial thread (notifies it about the completion of the request processing), which finally fetches the results from the `SearchRequest` and returns an appropriate `NetInf` message (step 18).

3.5.4 SearchServiceRDF

In our `NetInf` prototype we propose one implementation of the `SearchService` interface, the `SearchServiceRDF`.

This implementation uses a *SDB* database⁹ to store its data. *SDB* is a database which stores RDF data and supports SPARQL query processing on the data. It uses a SQL database as its back end.

The `SearchServiceRDF` implementation currently supports one query template for the template-based query specification. This template is used in scenario 2 (see Section 5.2). In this scenario we use **IOs** which represent shops and their inventory. With the template it is possible to search for shops which are near a position and offer some specific products. If one wants to use the template, one has to specify the position and radius around the position which mark the search area, and the products to search for.

⁹<http://openjena.org/wiki/SDB>

The SearchServiceRDF can be used in two operation modes. The mode it actually operates in is set via its configuration.

- If the SearchServiceRDF is connected to an **ES**, it can operate as a general global **SS** or specific global **SS** by only subscribing on **IOs** with specific characteristics (based on their attributes). In this way, it gets to know **IOs** of all **RSs** which publish to the **ES**. To store the information about these **IOs**, it uses a **RDFResolutionService** (see Section 3.4.2) which is operating on the same **NetInf** Node and storing the data in the **SDB** database which is used by this **SS**.
- If the SearchServiceRDF is not connected to an **ES**, it can be used to search in the data of a single **RDFResolutionService**. To do so, the **SS** and the **RS** have to use the same **SDB** database as their back end.

3.6 BO Handling

Besides the structured information that can be stored in **IOs**, there is another class of information that cannot be handled efficiently with **IOs** alone: BitlevelObjects (**BOs**) which may be music files or videos. This section describes how **BOs** are handled with respect to the **NetInf** architecture.

3.6.1 DataObjects

IOs, as they have been introduced in Section 3.1, cannot contain entire **BOs** because **IOs** are designed to contain only a limited amount of data. Thus, we introduced **DataObjects (DOs)** as a special form of **IOs**. Each **DO** represents and describes a **BO**. As mentioned before, an **IO** should only contain a small amount of data. So, to represent a **BO**, **DOs** contain one or more **Locators** pointing to copies of the actual **BO**. These **Locators** are implemented as attributes of the **DO** and refer to the actual **BO** which may be stored in a separate data store. Possible representations of such a **Locator** are HTTP links or identifiers of a file in peer-to-peer networks.

To be able to verify whether the downloaded **BO** is valid or not, the **DO** also has an attribute that contains the hash of the **BO**.

3.6.2 Transfer

In order to get the actual **BO**, the node architecture contains a so-called **TransferDispatcher** which manages several **StreamProviders**. Each **StreamProvider** is responsible for handling a specific protocol like **HTTP**. Think of a **StreamProvider** as an adapter for different types of data transfer protocols. Instead of first downloading the **BO** and then providing it to the requesting node component, the **TD** just creates a stream to the particular **BO**. This stream is of the type **InputStream** and can be handed over to other components like the **RESTful Interface** or the **Caching**. Therefore the validation of that **BO** has to be processed by the requestor. Validation within the **TD** happens only for chunks (see 3.6.3).

In order to influence the order in which the Locators are used to download the data, a `LocatorSelector` is used by the `TD` as shown in 3.10. This `LocatorSelector` sorts all available Locators by their priority to always get the best one for the user. Furthermore it provides the list of Locators as an `Iterator` so that the `TD` can move along to another source in the list if one suddenly fails.

The OpenNetInf prototype already contains StreamProviders for the HTTP and FTP protocols. Further protocols may easily be attached to the `TD`, provided the code to handle them is already in place. Take a look at the existing StreamProviders for an example of how to attach that code to the `TD`.

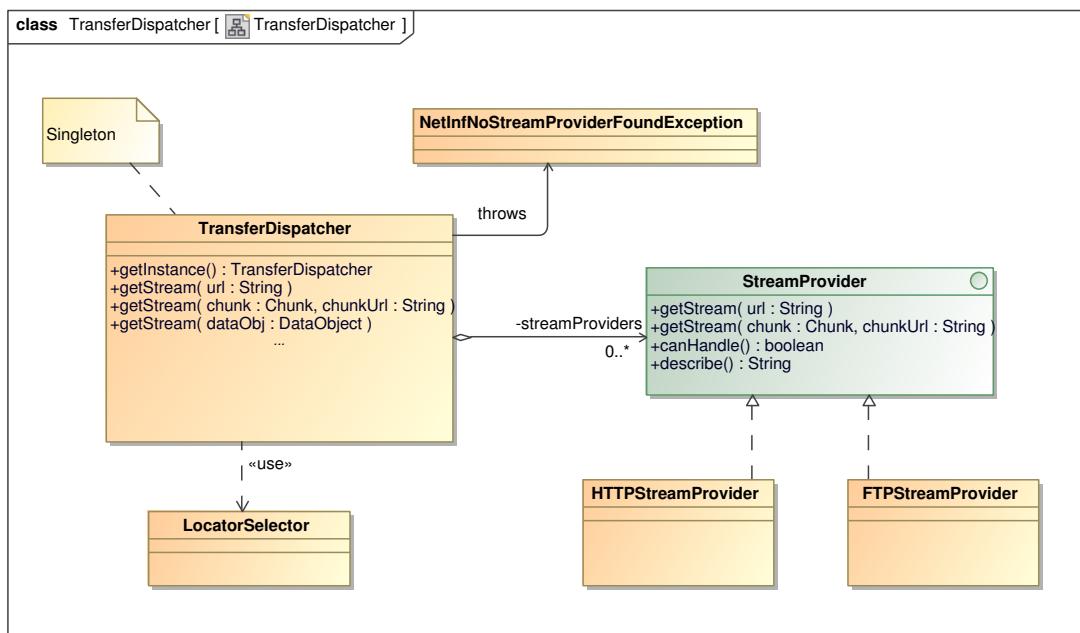


Figure 3.10: Overview of the TransferDispatcher

Note In the previous OpenNetInf prototype the transfer of `BOs` was handled by a `TransferService` of the `TransferController`. This was modified in order to reduce the complexity of the node architecture and to simplify the usage for nodes internally. The `TransferDispatcher` is now a solid part of the node component.

3.6.3 Chunking

In addition to the handling of whole `BOs`, the OpenNetInf prototype is able to handle single chunks of the `BO` as well. But the Chunking Concept of OpenNetInf is slightly different from the usual ones. Instead of splitting the regarding `BO` physically in several chunks and storing them afterwards, the `BO` is split conceptual in kind of ranges. This happens along with the Caching mechanism and can be switched off (`false`) and on (`true`) by this config line:

```
chunking = true
```

If chunking is enabled the caching component analyzes the **BO**, splits it in chunks of predefined sizes and determines the hash value of that chunks. A single chunk can then be requested by a HTTP Range Request. Since both the Peerside Cache and the Network Cache can handle such requests the OpenNetInf prototype can fully provide chunking for the cached **BOs**. Chunking can also be used along with external HTTP servers but only if they support Range Requests (supporting Range Requests is optional regarding the specification). Section 3.6.3.1 describes how the chunking concept is integrated into the data model and Section 3.6.3.2 and 3.6.3.3 how the chunks are merged by the TransferDispatcher. Two different approaches are implemented for that and it can be chosen between them.

Two of the big advantages of chunking are that chunks can be downloaded from multiple different sources and chunks can also be validated along with applications like video streaming. Without chunking the whole **BO** has to be downloaded at first before the hash can be checked.

3.6.3.1 Data Model Integration

When the chunks are processed for a particular **BO** the caching component determines the following parts:

- Number of chunk
- Hash value of the data of the chunk
- Total number of chunks

It is not necessary to save the offset and the length of the range, because the size of a single chunk is internally fixed (256 KiB) and can be calculated later with the number of the chunk. The data model has been extended by the attributes `chunks`, `chunk` and `hash_of_chunk` to store the necessary information. The example below shows how that looks like within a **IO**:

```
...
<netinf:chunks rdf:parseType="Resource">
<netinf:attributePurpose>SYSTEM_ATTRIBUTE</netinf:
    attributePurpose>
<netinf:attributeValue>Integer:50</netinf:attributeValue>
    <netinf:chunk rdf:parseType="Resource">
        <netinf:hash_of_chunk rdf:parseType="Resource">
            <netinf:attributePurpose>SYSTEM_ATTRIBUTE</netinf:
                attributePurpose>
            <netinf:attributeValue>String:7
                d4adf92ba3d22bb30884d6804d0acec86f90100</netinf:
                    attributeValue>
        </netinf:hash_of_chunk>
```

```

<netinf:attributePurpose>SYSTEM_ATTRIBUTE</netinf:
    attributePurpose>
<netinf:attributeValue>Integer:1</netinf:attributeValue>
</netinf:chunk>
<netinf:chunk rdf:parseType="Resource">
    <netinf:hash_of_chunk rdf:parseType="Resource">
        <netinf:attributePurpose>SYSTEM_ATTRIBUTE</netinf:
            attributePurpose>
        <netinf:attributeValue>String:
            d3b7d8609db48ffceb6f162376d286f5ae80b145</netinf
            :attributeValue>
    </netinf:hash_of_chunk>
    <netinf:attributePurpose>SYSTEM_ATTRIBUTE</netinf:
        attributePurpose>
    <netinf:attributeValue>Integer:2</netinf:attributeValue
        >
</netinf:chunk>
...

```

3.6.3.2 Sequential Chunk Stream

The first here described way of handling and merging chunks within the data transfer is called Sequential Chunk Stream. As the name says the single chunks of a **BO** are taken in a sequential order – visual described in Figure 3.11. The TransferDispatcher establish always only one HTTP connection to the chunk and downloads the chunks in their correct order. When a chunk is fully downloaded the hash of the chunk is validated and the chunk is pushed into the returned stream. In case of a fail the Sequential Chunk Stream tries another locator, if available, otherwise the whole stream fails.

3.6.3.3 Concurrent Chunk Stream

The second approach is called Concurrent Chunk Stream and deals in contrast to the previous technique with several concurrent HTTP connections. The chunks are downloaded by so-called BufferFillers in a non-sequential order, where the number of BufferFillers is flexible (currently set to seven). Figure 3.12 shows an example of the Concurrent Chunk Stream with four BufferFillers. Thus the first one handles chunk 0, 4, 8, the second one chunk 1, 5, 9 and so on. The particular intervals depend on the number of BufferFillers. To return the chunks as a stream with the correct order a collector iterates over the working BufferFillers in a Round-Robin way and gets the chunks in the correct sequence.

In that way this concurrent approach is much faster than the previous one and is therefore the default method in the prototype. To change that to the Sequential Chunk Stream the following line has to be changed to “2” in the TransferDispatcher Class:

```
private int chunkingTechnique = 1;
```

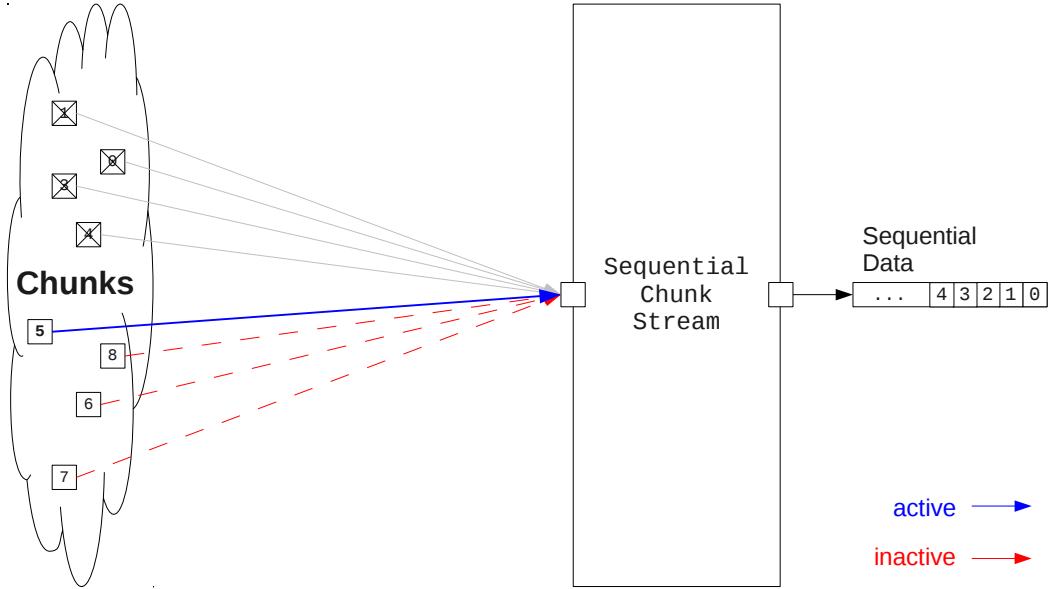


Figure 3.11: Sequential Chunk Stream

The validation of a single chunk takes place as well like described in the previous method.

3.6.4 Caching

Besides the caching of **IOs**, the node architecture provides caching of **BOs**, as well. The cache is implemented as a `ResolutionInterceptor` (see Section 3.4.4). The interceptor is triggered if a **DO** is requested. The caching implementation can then decide whether the **DO** with its corresponding **BO** should be cached or not. If the **DO** should be cached, the cache interceptor checks which of the installed cache modules should cache the **BO** of the requested **DO** and starts a threaded `CacheJob` for that. In case that the same **DO** is requested several times within a short time interval, the cache realizes that a `CacheJob` is running already. When the `CacheJob` is finished the related **DO** is put back to the Resolution Service with the new locators of the cached **BOs** (and the chunk lists, if enabled). Figure 3.13 shows how the caching component is build conceptually.

There are implemented two types of caches for OpenNetInf prototype: The Peerside Cache and the Network Cache. The caching component can handle multiple instances of both caches if necessary. But typically only one of each is needed.

Ehcach The underlying functionality of both caches is implemented by using Ehcache¹⁰ – an open source caching system. It is widely used and offers high flexibility and functionality. The Ehcache can be used with typical eviction strategies as well as with a persistent configuration.

¹⁰<http://www.ehcache.org/>

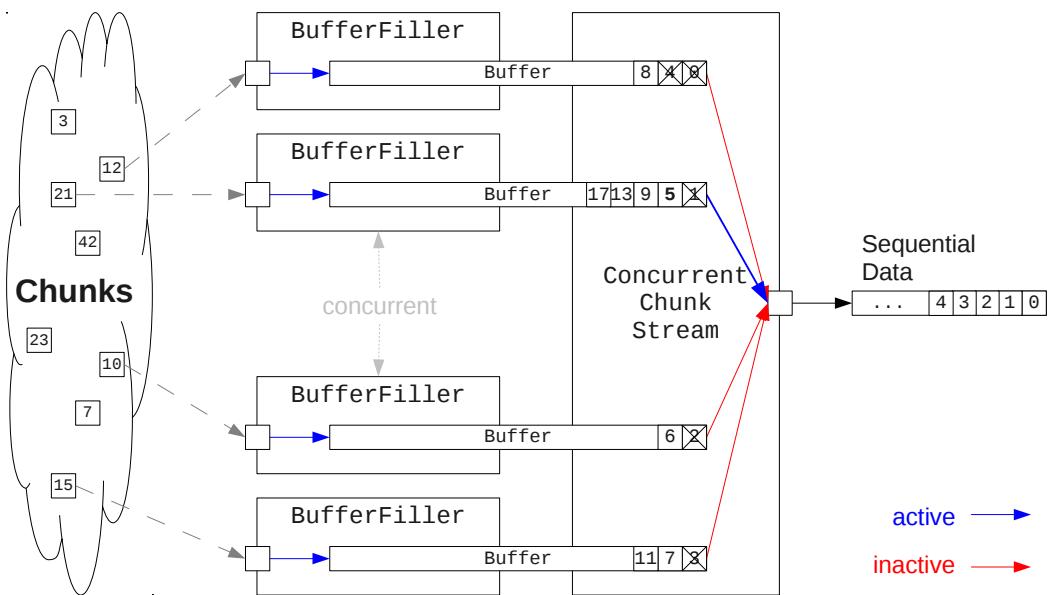


Figure 3.12: Concurrent Chunk Stream

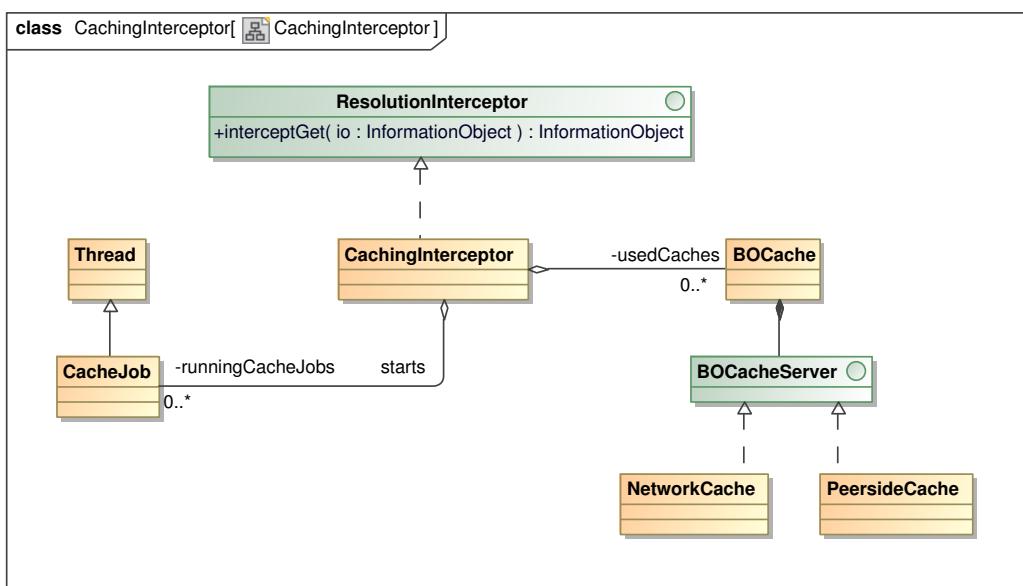


Figure 3.13: Caching Component

Furthermore it can be used in a embedded way like in the Peerside Cache and a standalone way like in the Network Cache. See more below.

Note The size of cachable files is limited to 2 GiB.

3.6.4.1 Peerside Caching

The Peerside Cache uses the Ehcache in an embedded way and provides an additional HTTP interface to make the embedded Ehcache reachable from outside. Thus the cached **BOs** can be registered as normal locators and downloaded from other clients. Since the Peerside Cache should be persistent and also provide the cached **BOs** after a node restart, the configuration which can be found in `configs/PeersideEhcacheConfig.xml` is set accordingly:

```
<cache name="PeersideCache"
      eternal="true"
      overflowToDisk="true"
      diskPersistent="true"
      maxElementsInMemory="1"
      diskExpiryThreadIntervalSeconds="1"
/>
```

Conceptual the Peerside Cache is made for caching on the client side. Even running several Peerside Caches in parallel is possible. In context of a **MDHT** scenario it can be used to enable caching for client nodes which are connected to access nodes of the **MDHT** system. Therefore all nodes on the first level will see the Peerside locators, if configured like following:

```
# PeersideCache
peerside.cache.numberOfCaches = 1
peerside.cache.0.access.host = localhost
peerside.cache.0.access.port = 11080
peerside.cache.0.scope = 1
```

The `peerside.cache.X.scope` property defines up to which **MDHT** level the locators are visible. `peerside.cache.numberOfCaches` states how many Peerside Caches should be used – the necessary properties have to be adjusted accordingly.

Note The Peerside Cache replaces the old BOcaching Component (`netinf.node.resolution.bocaching`) which is marked as deprecated now. If you are using this cache currently we strongly recommend switching to the Peerside Cache.

3.6.4.2 Network Caching

The Network Cache uses the Ehcache as an external service and communicates with it via the RESTful Interface of the Ehcache. Therefore it is necessary to setup at first a dedicated Ehcache standalone server which offers such mechanisms. You can find on

the ehcache.org¹¹ website how to install, run and configure the server. Thus the Network Cache does not use an embedded caching system like the Peerside Cache but operates via HTTP messages with the Ehcache server to store BOs.

Along with the development of the MDHT Resolution System the Network Cache was developed to enable caching within the MDHT topology/network. Thus, a single NetInf Node can be attached to several Network Caches, where these caches are assigned to the respective level of the MDHT. The property network.cache.X.scope specifies that assigned level. The propagation of the cached locators is also limited to that level and allows caches to be seen only from nodes that are part of this level. Nodes outside of this level are not able to see cached locators of this Network Cache. Below you find a sample configuration for one Network Cache that is assigned to the second level:

```
# NetworkCache
network.cache.numberOfWorkers = 1
network.cache.0.address = agk-1pc28.cs.upb.de
network.cache.0.port = 10180
network.cache.0.tablename = /ehcache/rest/netinf
network.cache.0.scope = 2
```

The basic idea of the Network Caches is that individual network providers could run several caches within their own domain in order to reduce inter-domain traffic. That means that a requested file has to be transferred only once and can be provided from that on by the own set of Network Caches. Another advantage is that users get a nearer copy of the file and the originally source becomes no bottleneck.

3.7 Security

One major part of the concept of NetInf is that we *do not care wherefrom some piece of information is received*. This shall also work in case of confidential data, or data that we need to rely on.

In more detail: We want to store data in NetInf, that we need to rely on. For example, we want to store the information whether or not a certain kind of mushroom is poisonous. If someone is just about to eat such a mushroom, he or she certainly needs to rely on the information that it is not poisonous. Thus, even if the information was received from a node inside NetInf, that we do not trust, we want to know whether the information was modified compared to when it was stored in NetInf. In other words, we want to ensure that the information is still the same as entered by the one who stored the information in NetInf. We could then decide whether or not we trust in the information given by a certain person – the one who stored the information in NetInf. So we want to ensure integrity of data: We decided to use signatures, as, for example, discussed by Dannewitz et al. [3]. Our approach to data integrity is discussed in Section 3.7.1.

Furthermore, we want to store confidential data somewhere in NetInf. Maybe, we even want to store that data at a NetInf Node we do not trust. However, that data shall of

¹¹<http://ehcache.org/documentation/user-guide/cache-server>

course remain confidential. Thus, we decided to encrypt confidential data. Our approach to encryption is discussed in Section 3.7.2.

All the security functionality that is described in this section, is part of the `netinf.common` library (see Section 3.2).

Security Manager To allow easy integration of our security related approaches into the `NetInf` architecture, we designed the `SecurityManager` interface. It provides the two methods `checkOutgoingInformationObject` and `checkIncomingInformationObject`. The term *outgoing* is related to `IOs` in a *put* operation. From the point of a `NetInf` enabled application, an `IO` is *outgoing* (i.e., it leaves the application) when it is stored to `NetInf`. The term *incoming* is, thus, related to `IOs` in a *get* operation. When a *get* operation is performed, the `IO` that arrives at the application is an *incoming* `IO`.

`checkOutgoingInformationObject` first creates signatures to allow integrity verification, and subsequently encrypts confidential data. This has to be done just before the `IO` is sent to a remote `NetInf` Node. Thus, this method is integrated in the `RemoteNodeConnection`'s *put* operation (see Section 3.2.4). Just before the *put* is performed, `checkOutgoingInformationObject` also verifies the correctness of the `IO` and stops the *put* if the security related information of the `IO` is already incorrect – for whatever reason.

`checkIncomingInformationObject` first decrypts encrypted parts of the `IO`, and subsequently verifies signatures to data integrity. This has to be done directly after the `IO` is received from a remote `NetInf` Node – otherwise the `IO` would possibly still contain encrypted data and the `NetInf` enabled application could not interpret the `IO`'s content. Thus, this method is integrated in the `RemoteNodeConnection`'s *get* operation. The integration of the `SecurityManager` to the `RemoteNodeConnection` is illustrated in Figure 3.14.

So far, `IOs` created by `NetInf` enabled applications are provided with security related information (attributes are encrypted, signatures exist). Furthermore, `IOs` received by `NetInf` enabled applications are verified for those pieces of information (attributes are decrypted, signatures are verified). Thus, a receiving `NetInf` enabled application would recognize modification of data.

However, an attacker may modify an `IO` just after it leaves a `NetInf` enabled application. The `NetInf` Node where this `IO` is subsequently stored at, would store the *modified* `IO`. The `NetInf` Node would then respond to *get* requests, sending a *modified* `IO` instead of the `IO` the `NetInf` enabled application *originally* wanted to store in `NetInf`. Although the receiving application will recognize the modification, this operation takes additional time that could have been used to look for an unmodified version of the `IO` instead. Thus, `NetInf` Nodes use `checkOutgoingInformationObject` and `checkIncomingInformationObject` as well.

Since it is the `ResolutionController` interface of the `NetInf` Node (see Section 3.4.1), that handles the *get* and *put* operations, it is also the `RC` to perform `checkOutgoingInformationObject` and `checkIncomingInformationObject`. `checkOutgoingInformationObject` is part of the `RC`'s *put* method,

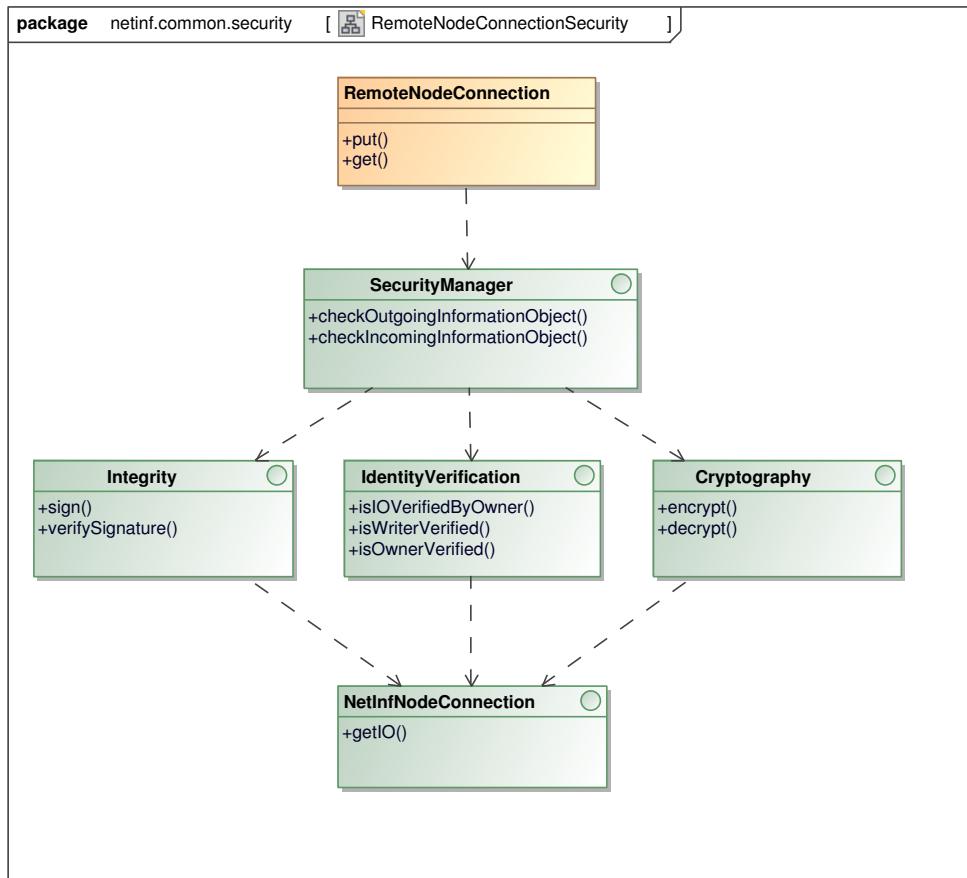


Figure 3.14: A NetInf enabled application uses the `RemoteNodeConnection` to get and put `InformationObjects`. The `SecurityManager` is integrated in these methods to automatically apply our security approaches. The classes implementing our security approaches may themselves use `NetInfNodeConnections` to get further `InformationObjects` they need to perform security checks.

`checkIncomingInformationObject` is part of its get method. Figure 3.15 illustrates this integration.

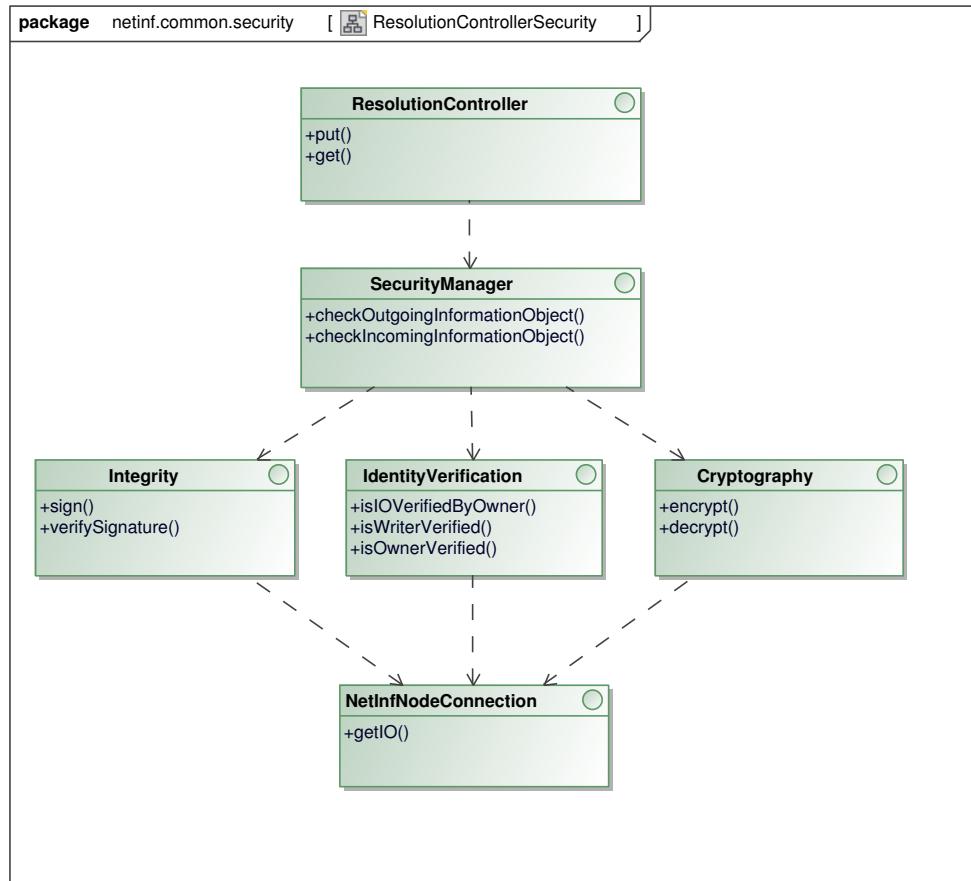


Figure 3.15: The NetInf Node’s `ResolutionController` gets and puts `InformationObjects`. The `SecurityManager` is integrated in these methods to check for unauthorized modification of data as soon as possible.

According to the arguments given so far, the `RC` would have to perform the “plausibility” checks of `checkOutgoingInformationObject` and `checkIncomingInformationObject` only. After all, the `NetInf` enabled applications already encrypted and signed the `IO`, so why should the `NetInf` Node’s `RC` do? In fact, why should some `NetInf` Node even be able to sign an `IO` on behalf of someone else?

In case of `NetInf` enabled applications, the `NetInf` Node actually does not perform anything else than these plausibility checks (to observe modifications as soon as possible). However, in case of applications that directly access a `NetInf` Node, without using the `netinf.common` library, the `NetInf` Node may be provided with someone’s identity to sign and verify signatures, and to encrypt and decrypt `IOs` on behalf of that someone. Thus, applications may even benefit from `NetInf`’s security approaches without actually integrating the `netinf.common` library into that application. However, since, instead of the application, the `NetInf` Node creates and verifies signatures, and

encrypts and decrypts, the communication between application and **NetInf** Node needs to take place in a trusted environment. In case of a man in the middle attack between application and **NetInf** Node, the attacker would have access to unencrypted information and might modify attributes before signatures are calculated.

3.7.1 Integrity

When an **IO** is retrieved subsequently to a get operation, we expect to get the **IO** we asked for, not a different **IO**. However, it is not guaranteed that we actually received the **IO** we asked for. For instance, there might have been an attacker modifying the **IO** before we receive it, as illustrated in Figure 3.16.

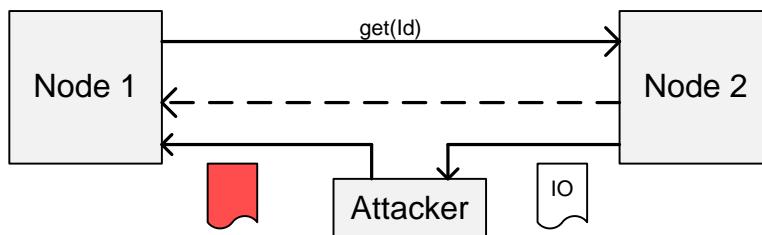


Figure 3.16: An attacker may modify an **InformationObject** that is sent over an unsecured channel, in response to a get request.

The *Integrity* functionality of the *Security* addresses the integrity of **IOs**: unauthorized modification of certain parts of **IOs** shall be detectable.

To detect modification of an **IO**, we build on signatures using a Public Key Infrastructure (**PKI**). Our **NetInf** prototype stores a particular kind of **IOs** – the so-called **IdOs**. **IdOs** represent identities such as persons. In his or her **IdO**, the public key of the person is stored. Obviously, the private key should be known to the owner of the identity only.

Now, the basic idea of our *Integrity* functionality is to build a signature of an attribute using the private key, and then put that signature in the **IO** as well. Furthermore, the reference to the modifier's **IdO** is provided. Thus, whoever receives the **IO** is able to verify, whether the attribute was modified since the signature was built, because the according public key is available at the **IdO**.

Furthermore, it needs to be defined who is allowed to change an **IO** at all. This is done by a list of **IdOs** of users authorized to change the **IO**. This list must not be changed by anyone else than the owner of the **IO**. Therefore, we include a hash of the owner's public key in the identifier of the **IO**. Since the identifier is known to the user, as soon as he or she asks for the **IO**, he or she may then verify, whether the list of authorized users has been signed by the **IO**'s owner. This relation between identifier and list of authorized users is discussed in more detail in the *Identity Verification* paragraph.

Detect Modifications To detect all kinds of modifications to an attribute, the signature must not be built using the attribute's value only. The signature needs to be built using all the information of an attribute: value, identification, etc.

However, even if the signature is built using all this information, the attribute and its signature may be moved to a different location inside the **IO**, as demonstrated in Figure 3.17, or even to a completely different **IO**. For instance, we care about attribute **A**, which originally was a subattribute of attribute **B**. Person **P** built the signature for **A** and attached it as a subattribute to **A**. An attacker may now move the attribute **A** to be a subattribute of attribute **C**. The signature of **A** would still be valid. Indeed, value and identification of **A** have not been modified, but the position of **A** has been changed. Thus, the signature has to be built using the position of the attribute as well.

So if the signature is built using the identifier of the **IO**, the path down to the attribute, the identification, and the value, neither the attribute itself, nor its position may be modified. We call this the *attribute content string*.

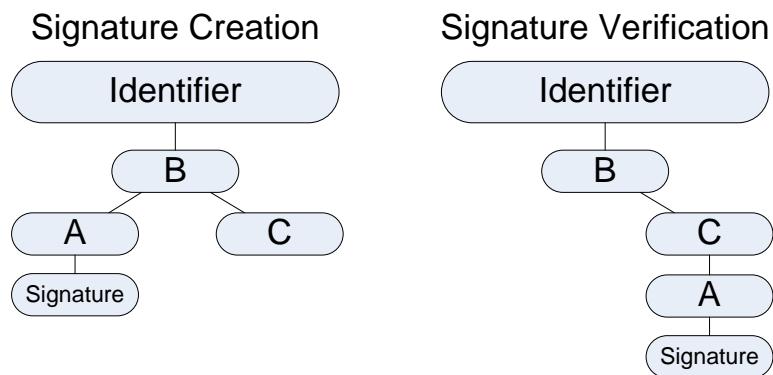


Figure 3.17: When the signature of an attribute is built, the position (identifier and path) of the attribute has to be considered. Otherwise, the attribute and its signature may be moved to a different position, and this would not be detectable while signature verification.

Detect Drop Although we are able to detect modifications of present attributes, we are not yet able to detect *missing* attributes. An attacker may simply drop an attribute and its signature, and whoever receives the **IO** is not able to detect that the attribute is missing.

To enable such a “drop detection”, we introduced an additional concept of *overall signature*, which is an **IO** wide signature. The idea is to mark certain attributes to be *secured in overall* – they get an `is_secured_in_overall` subattribute – and to calculate one signature of the concatenated *attribute content strings*.

If this *overall signature* is mandatory, an attacker may neither drop any of the marked attributes, nor modify them. As Figure 3.18 illustrates, the signature is built using all marked attributes. If one of them is modified (e.g., dropped), the signature will not be verified successfully.

Detect Add Please pay attention to the fact that so far an attacker may arbitrarily add additional attributes. On the one hand, the attacker may add attributes without signature.

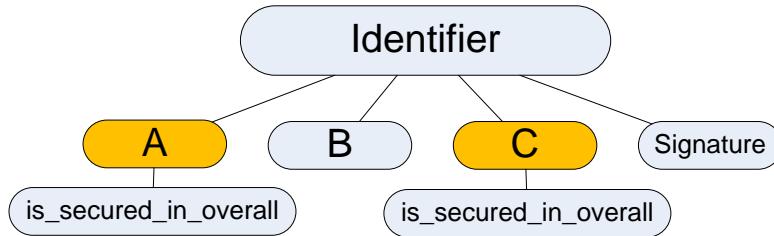


Figure 3.18: In case the overall signature is mandatory, attributes included in the overall signature may not be dropped. Otherwise, the signature will not be verified successfully.

Unsigned attributes may be used for additional information you do not necessarily have to rely on. For example, a **NetInf** Node may add additional locators of a binary object as unsigned attributes. After downloading the binary object, the correctness of that binary object can be verified using the hash of the file. The hash itself has in fact been stored to the **IO** as a signed attribute – and thus is not changeable by an attacker. To avoid random add of unsigned attributes, it is conceivable to add a mandatory attribute to the overall signature, indicating whether or not the **IO** may contain attributes without signature.

On the other hand, the text above does not yet explain, why an attacker (i.e., someone who is not allowed to change an **IO**) may not add a signed attribute using his or her own **IdO** and public/private key pair. Such a signed attribute would give the impression that the attacker was allowed to change the **IO**. This problem is taken care of by the so-called *Identity Verification*.

Identity Verification Besides verifying signatures of attributes, it also has to be checked, whether or not someone is allowed to add or modify attributes of an **IO** at all, independent of the validity of the signature. Even if the signature of an attribute is valid, maybe the person who added the attribute was not even allowed to add the attribute. So *Identity Verification* aims at *Rights Management*.

The basic concept is to provide a list of **IdOs** whose owners are allowed to change an **IO** – the so-called *Authorized Writers*. This list of **IdOs** is provided as attributes in the **IO**, signed by the **IO**'s owner. We want the owner of the **IO** to be the only one to decide who may change the **IO**. So the owner shall be the only one to define the *Authorized Writers List*.

When the **IO** is created initially, one part of its identifier is the *hash of public key* (HASH_OF_PK) identifier label, with the hash of the creator's public key as its value. Furthermore, the **IO** gets an **owner** attribute containing the identifier of the creator's **IdO**. Thus, when an **IO** *I* is retrieved, we may also get the **IdO** using the **owner** attribute, and check whether the contained public key belongs to the HASH_OF_PK identifier label of *I*. If so, we know we have the owner's public key and may validate the signatures of the *Authorized Writers List*.

Please note that at the beginning, the creator of the **IO** is its owner. So far, there is no convincing concept how to change the owner of an **IO**. This open issue is discussed in detail in Section 8.2.1.2.

By now, we know whether or not the owner of a certain **IdO** is allowed to change the **IO**. Subsequently, the *Identity Verification* simply walks through all the attributes and checks whether the `writer` attributes – indicating who signed the attribute – refer to **IdOs** contained in the *Authorized Writers List*. Figure 3.19 illustrates that identity verification recognizes the writer of attribute **B** was not authorized to modify the **IO**.

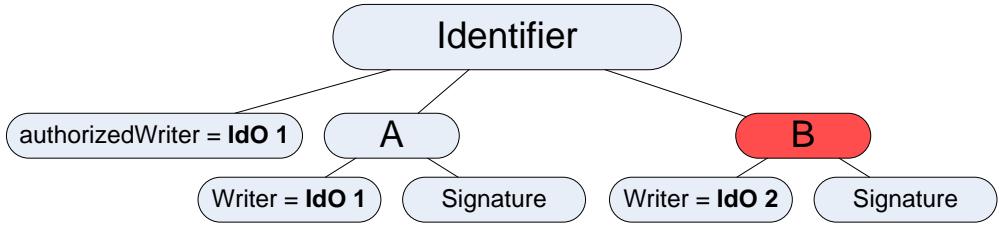


Figure 3.19: The writer of attribute **A** is part of the *Authorized Writers List*. The writer of attribute **B** is not. Thus, attribute **B** is not provided with a valid signature, since the writer is not allowed to modify the **InformationObject**.

Details The explanation above already introduces the two types of subattributes `writer` and `signature`. The `writer` subattribute refers to the **IdO** responsible for the signature (i.e., who added or changed the signed attribute). The signature itself is given in the `signature` subattribute.

Actually, the `writer` subattribute is not only designated to contain the identifier of an **IdO**, but rather to contain a path to a certain attribute in the **IdO**. This allows several so-called subidentities to be contained in one single **IdO**. For each subidentity there would be a separate public/private key pair.

Furthermore, there exists an additional subattribute below the `signature` attribute: `signature_identification`. The `signature_identification` attribute refers to the algorithm that has been used to build the signature. Thus, new signature algorithms may be introduced to **NetInf** easily.

Outgoing IOs When an **IO** is sent to a **NetInf** Node because of a put operation, the security manager checks for the occurrence of `writer` attributes without related `signature` attribute. This constellation is interpreted as a request for building the signature. The security manager calls the `sign()` method of the `Integrity` interface to build the requested signature.

Incoming IOs Whenever an **IO** is received, first of all the security manager checks whether the *Authorized Writers List* has been signed by the owner.

Afterwards, the security manager checks for the occurrence of `signature` and `writer` attribute. This combination is interpreted as a present signature. Accordingly, the security manager calls the `isSignatureValid()` method of the `Integrity`

interface to verify the signature. The security manager also calls the `isWriterVerified()` method of the `IdentityVerification` interface to check whether the one who signed this attribute was allowed to modify the `IO` at all.

Invalid signatures and unauthorized modifications are marked in the `IO`, appending certain subattributes specifying the error.

3.7.2 Cryptography

In an information-centric network there are no trusted nodes to control access on information, but users may want to make information available only for certain readers. Because the access on data (that is distributed) is not restrictable, the information needs to be published in a way so that it can only be read by the desired identities.

The *Cryptography* functionality of the *Security* addresses the readability of `IOs` and attributes. This is done by encrypting the content with a key only the readers have access to. Since every `IdO` has at least one public/private key pair, content may be encrypted asymmetrically using the public key, which is available to everyone. Then only someone who knows the corresponding private key can read the content afterwards; this should be the owner of the identity only. However, encrypting the whole content of an `IO` asymmetrically once for every reader is very time-consuming. Instead, the content is encrypted symmetrically using a generated key which is then itself encrypted asymmetrically as described before once for every reader.

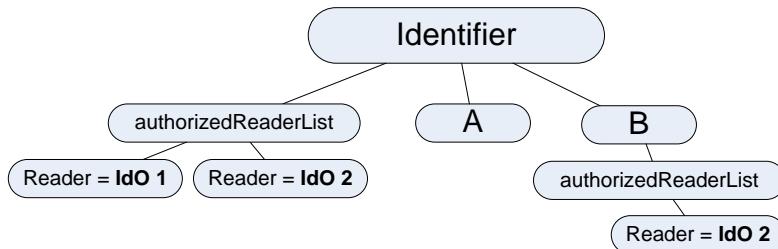


Figure 3.20: An `InformationObject` with *Authorized Readers List* for the `InformationObject` and attributes

Outgoing To restrict readability of an `IO` or an attribute, a *Authorized Readers List* has to be added to the `IO` or attribute as a subattribute. This *Authorized Readers List* has to contain one subattribute for every identity supposed to be able to read the content after encryption. Figure 3.20 shows an `IO` with two attributes. The `IO` is only readable for identity 1 (`IdO 1`) and identity 2 (`IdO 2`). Its attribute A is readable for everyone able to read the `IO` and attribute B is only readable for identity 2. `IOs` and attributes containing a *Authorized Readers List* are automatically encrypted when they are transferred using the `encrypt()` method of the *Cryptography* interface. Subattributes of `IOs` and attributes may have more restrictive *Reader Lists* than their parents, thus access to subattributes may be more restrictive than the access to their parents. Encryption is done

bottom up. This means IOs and attributes are encrypted only if they have no further unencrypted subattributes containing a *Authorized Readers List*. Figure 3.21 shows the IO from Figure 3.20 after the first step of encryption. When all such subattributes are encrypted their parents no longer have such a subattribute so they are then encrypted as well. Figure 3.22 shows the encrypted version of the IO.

Encrypt() The content with restricted readability is first serialized using the `serializeToBytes()` method of the `NetInfoObjectWrapper` (see Section 3.1.2). The serialized version is then encrypted and added as an `encrypted_content` attribute to the IO. This attribute contains the encrypted content as a string, a subattribute stating the algorithm that was used for encryption, and a list (`ReaderKeyList`) containing the different asymmetric encryptions of the generated symmetric key – one for each reader (figures 3.21 and 3.22). The identifier of this subattribute is the path of attributes in the IdIO to the public key that was used to encrypt the generated key. The value of this subattribute is the encrypted generated key as a string. If an attribute was encrypted, the new `encrypted_content` attribute replaces this attribute and takes its position in the parent IO. If the encrypted content was an IO, the `encrypted_content` attribute is added to an empty IO with the identifier of the old IO and it is marked as an encrypted IO. The attribute was encrypted first.

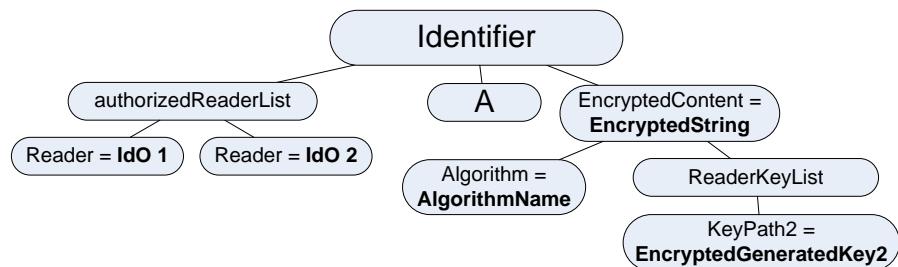


Figure 3.21: The InformationObject from Figure 3.20 with attribute B being encrypted

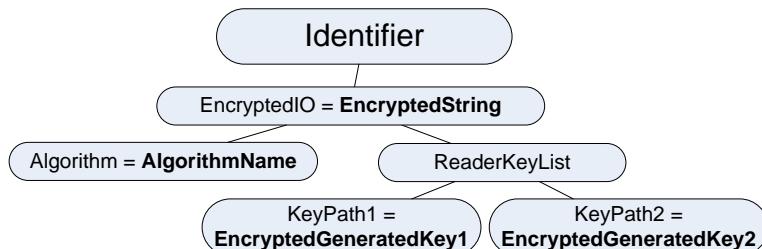


Figure 3.22: The InformationObject being completely encrypted

Incoming Encrypted IOs (and IOs containing encrypted attributes) are automatically decrypted when they arrive – if the required private keys are available. Every

`encrypted_content` attribute is decrypted using the `decrypt()` method of the Cryptography interface, starting at the top. If the whole **IO** was encrypted, it is the **IO** that is decrypted first. The current **IO** is then replaced by the decrypted **IO**. Afterwards, the whole **IO** is recursively decrypted, in a top to bottom approach. In case there is an `encrypted_content` attribute the private key is known for, the attribute is decrypted and the `encrypted_content` attribute is replaced by the decrypted attribute. Subsequently, this is recursively done for subattributes. Thus, to decrypt subattributes an identity needs to be able (i.e., allowed) to read their parent attributes and the **IO** if they are encrypted as well.

Decrypt() To decrypt an `encrypted_content` attribute, the algorithm and the key used for encryption are required. First, the key is retrieved by reading the list containing asymmetrically encrypted versions of the generated key. For every path to a public key found in the list it is checked whether the respective private key is available. With the help of this private key the generated key is decrypted. Then, the algorithm that was used to encrypt the content is retrieved by reading the corresponding attribute. Finally, knowing the required information the content is decrypted.

3.8 Event Service

This section describes the Event Service (**ES**) used within the project group. We assume familiarity with the concepts of event services. To gain additional information have a look at the introduction given in [5]. In general, the **ES** might be used to solve two problems:

1. Decoupling of different services provided by the nodes within **NetInf**.
2. Update of client applications that are situated on top of **NetInf**.

In the former case, the **ES** provides a decoupling of the search from the storage of the **IOs** since the search can be informed asynchronously about changes in the space of **IOs** and can accordingly update its index. In the latter case, client applications can be informed about changes of **IOs**, resulting in the possibility to create context- and location-aware services. An example application is the ShoppingTool which is described in Section 5.2.2.

The overall goal was to create an adaptable **ES** that can solve both needs with the help of currently available **ES** implementations like *Siena*¹² or *Hermes* [10]. Therefore, we developed an Event Service Framework (**ESF**) that might be extended in order to use arbitrary **ES** implementations for different purposes.

The necessity to have such a generic framework that can be extended by several specific currently available implementations of **ESs** results from several factors. First and foremost, every implementation of an **ES** distributes the messages, like events and subscriptions, in a particular fashion and uses different algorithms for routing and matching

¹²<http://serl.cs.colorado.edu/~serl/dot/siena.html>

these messages. Usually, these algorithms are powerful in some defined areas, and less effective in others. Thus, for example, *Siena* is powerful in distributing large amounts of events in the case of a relatively small amount of subscriptions. This comes in handy, when distributing events for the **SSs**, where only a relatively small amount of **SSs** exists, which are interested in a large amount of events. On the contrary, *Hermes* is able to handle a large amount of subscriptions which claim interest in a relatively small and defined amount of events. Accordingly, this implementation is suitable to fulfill the requirements for distributing events for the client applications. Recall that millions of client applications might exist, where each claims interest only in a small amount of events.

Additionally, the generic **ESF** provides the possibility to switch to another **ES** implementation without having to modify the **NetInf** Nodes. The **ESF** is an additional indirection between the nodes and the concrete implementation of the **ES**. Thus, if the concrete **ES** implementation changes then only the **ESF** has to be adjusted, but the nodes can remain unchanged.

Finally, we have added some functionalities to the **ESF** which are necessary within an information-centric network. This implies that they are automatically available independent of the concrete **ES** implementation.

This section describes the **ESF** and one binding of a currently implemented **ES** – the *Siena* **ES** – to this **ESF**. It starts with the illumination of the **ESF**.

3.8.1 Event Service Framework

The main idea of the **ESF** is to stay independent of any already implemented **ES**, which might be used for the distribution of events and subscriptions. Unfortunately, not every **ES** provides a defined interface for standardized languages like XML. Therefore, the methods of the **ES** cannot be invoked only with the help of XML messages. One aspect that is common for all **ESs** is that they have an API in their respective programming language. Accordingly, the **ESF** has to be implemented in the same programming language like the underpinned **ES** that is used by the **ESF**. It is conceivable to have different **ESFs**—one for each programming language—so that any **ES** implementation may be used no matter which programming language it is written in. The typical tasks which are then performed by the **ESF** is the setup of the implemented **ES** and the translation from the **NetInf**-specific messages to the target format of the **ES**. The following paragraphs describe the implementation of the **ESF** in Java, which was chosen because *Siena* is implemented in Java as well.

3.8.1.1 Architecture of the ESF

We start with the description of the **ESF** by having a look at how the **ESF** fits to the whole concept of an information-centric network and how it relates to the concepts of **NetInf** Nodes and their provided services.

Physical View The physical view depicted in Figure 3.23 shows a typical communication and distribution structure of the peers within an information-centric network that is connected to an **ES**. Additionally, it shows which components are active on the different peers.

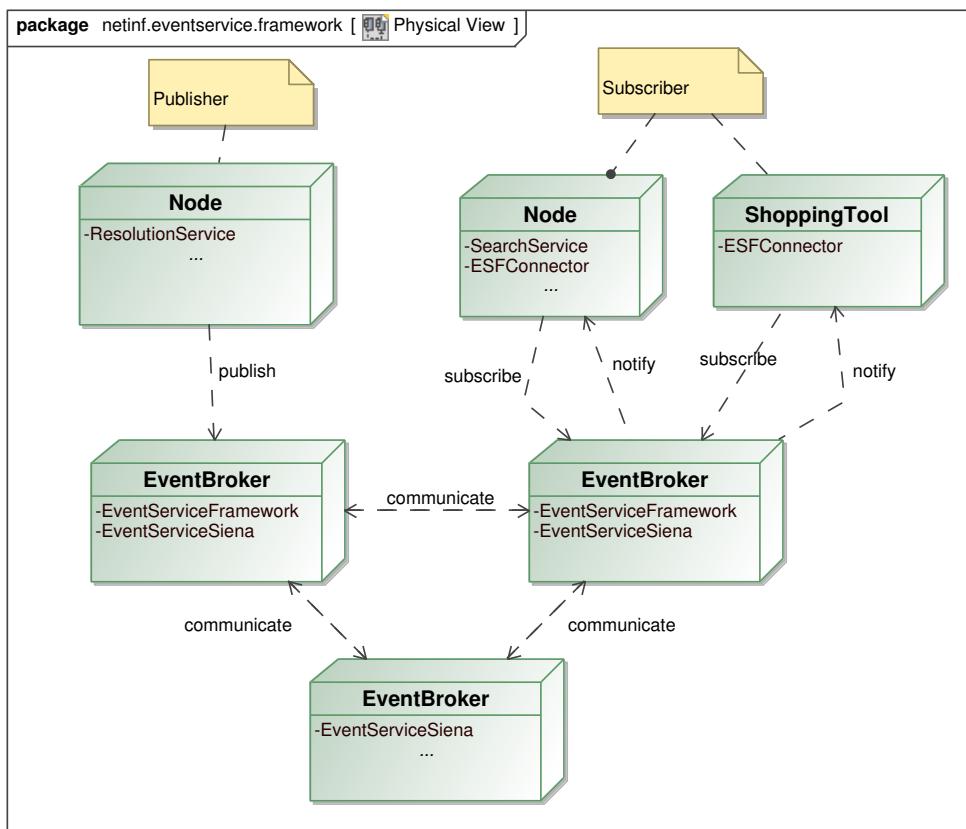


Figure 3.23: Physical view of the ESF

The **ES** consists of three different event brokers. All event brokers together realize the **ES Siena**. The communication between these event brokers depicted in Figure 3.23 is the internal communication typical for *Siena*. Only two of the three event brokers are equipped with the **ESF** component. This means only these event brokers can be connected to peers of our information-centric network, like, for example, nodes and **NetInf** Applications. The **ESF** can be seen as an intermediary between our information-centric network and the *Siena* implementation of an **ES**. Accordingly, it has to be situated at the edges of the **ESs**.

Nodes can act as publishers of information, as shown in the upper left corner. Every time an **IO** is created, deleted or updated, the delta – this means the old and the new version of the **IO**, if present – is published to the associated event broker, running the **ESF**. The **ESF** translates this event into the target representation of events (in case of *Siena* called **Notifications**) and publishes them directly within the native implementation of the **ES**.

On the other hand, nodes and applications can act as subscribers. This means they can receive matched pieces of information in the form of a delta of states of **IOs**. In order to receive these pieces of information, they have to be connected to an event broker that runs the **ESF** component. This is achieved with the help of the **ESFConnector** which encapsulates the communication to the remote **ESF** component.

Similar to the publisher-case, the **ESF** represents an intermediary, which is responsible of translating the native events to the typical event representation within **NetInf**, consisting of two states of the same **IO**. Two concrete examples for subscribers are shown within the figure. The **SS** might update its index according to the events received from the **ES**, and the **ShoppingTool** application might update the products of its loaded shopping list, according to the pieces of information provided by the **ES**.

Class View The class view on the **ESF** contains the most important classes and their most important methods (only names, no signatures) which are required to gain an insight into the sequence of actions in the **ESF**. It is depicted in Figure 3.24. The description starts at the top of the figure and continues downwards.

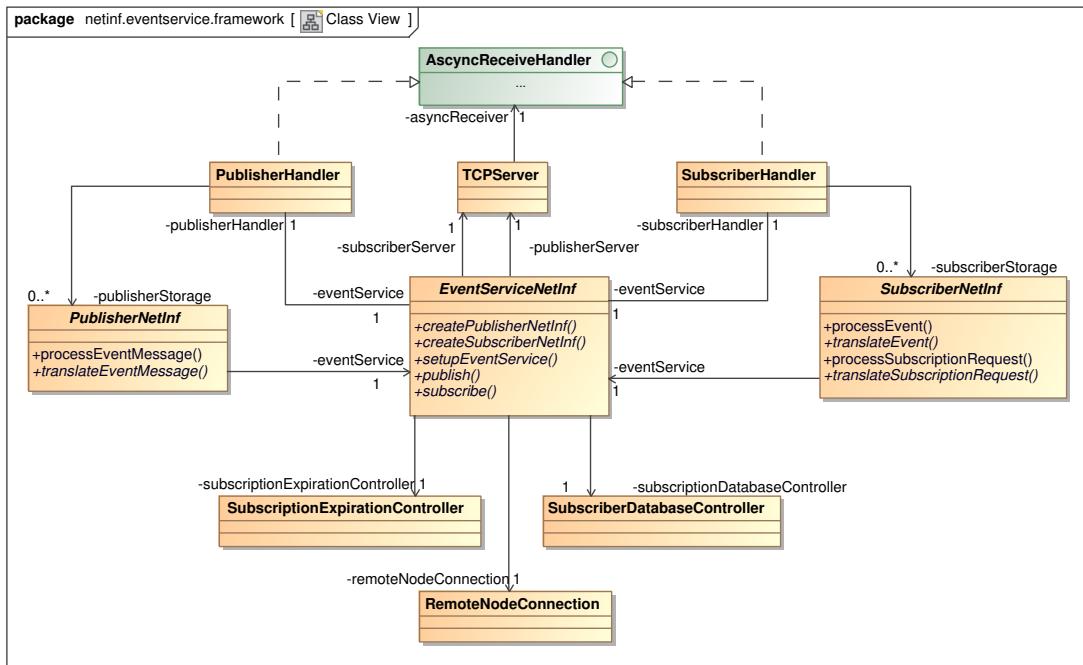


Figure 3.24: Class view of the ESF

As already mentioned, the **ES** provides two different pieces of functionality, which are *informing about changes* and *being informed about changes*. These two different pieces of functionality are represented and can be accessed via two instances of the class `TCPServer`. They represent simple TCP servers which are accessible on a predefined port. Each server delegates the incoming messages – instances of the type `NetInfMessage` – to the appropriate handlers. The handlers know how to process the incoming messages, and where to delegate the task. They maintain the set of publishers and subscribers.

The three classes in the middle are the most important ones, they provide the extensibility of the **ESF**. They are the abstract classes `PublisherNetInf`, `EventServiceNetInf`, and `SubscriberNetInf`. These three classes have to be extended for each different implementation of an **ES** that should work with `NetInf`. They basically encapsulate the logic which is required for processing events and subscriptions. The only missing functionality is the actual translation of the events and subscriptions from `NetInf` (called `EventMessage` and `ESFSubscriptionRequest`) to the target events and subscriptions of the **ES**. This functionality is implemented within the according subclasses, which are specific for one implementation of an **ES**. The collaboration of these three classes is depicted in the following paragraph, named *Operational View*.

The instance of the type `EventServiceNetInf`, a singleton within one running **ESF** component, is the linchpin of the **ESF**. It is responsible for creating the publishers and the subscribers, maintaining a connection to the servers, and incorporating some additional functionalities.

These functionalities provided by the **ESF** are automatically added to each implementation of an **ES**. They make the **ESF** even more attractive. They include the possibility of timed subscriptions – which means that subscriptions have a defined validity period and expire automatically –, storage of the subscriptions within a local database – which makes the node more reliable (compare paragraph on local reliability) –, and storage of events within an information centric network. These three functionalities are realized by the three classes `SubscriptionExpirationController`, `SubscriberDatabaseController` and `RemoteNodeConnection`.

Operational View The two overviews presented in this paragraph describe briefly the general sequence of actions performed by the **ESF** in case a publisher publishes an event, or a subscriber sends a subscription and receives a response. These overviews are shown in Figure 3.25 and Figure 3.26.

The two given sequence diagrams contain objects of some non-abstract subclasses of `SubscriberNetInf`, `EventServiceNetInf`, and `PublisherNetInf`. Furthermore, only the pieces of information are shown which are necessary to convey the general sequence of steps of the **ESF**. Several intermediate calls, and complex signatures of methods were omitted for the sake of readability. In both cases, it is assumed that the publisher and the subscriber connect to the **ESF** for the first time.

At first, the publisher side is described. In the depicted Figure 3.25, the publisher sends an instance of the type `EventMessage` to the port where the TCP server of

3 Components in Detail

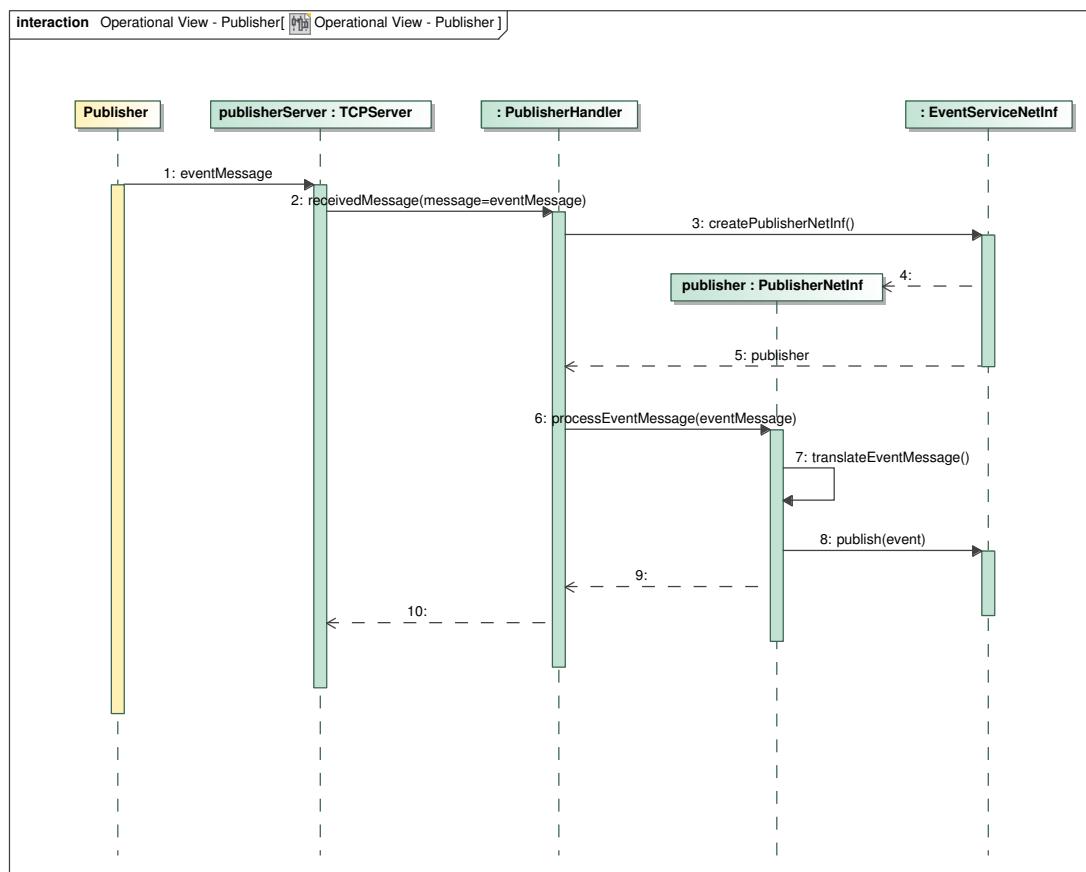


Figure 3.25: Operational view of the ESF

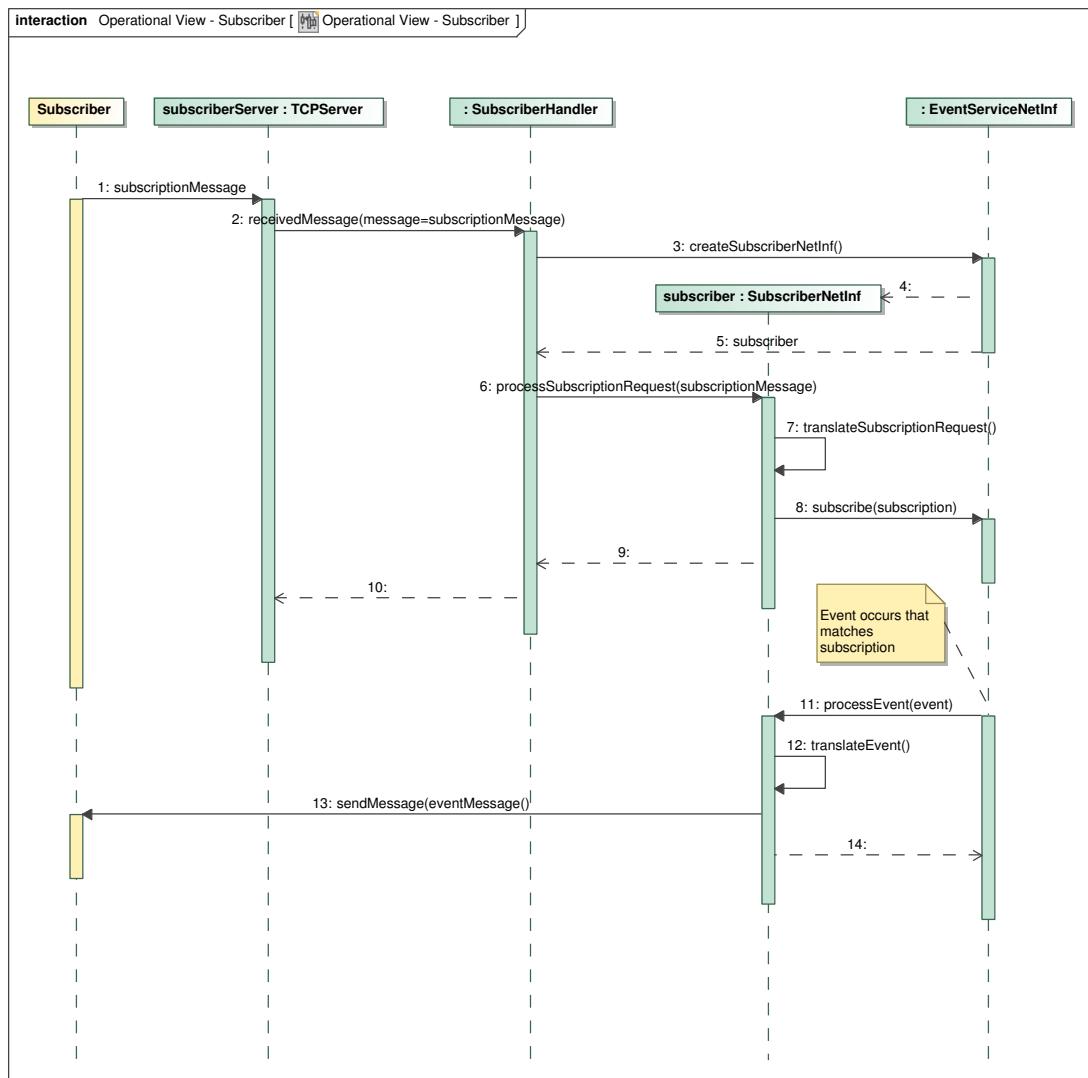


Figure 3.26: Operational view of the ESF

the **ESF** is listening for event messages. With the help of a handler, which recognizes that the publisher is connected for the first time and which triggers the creation of a corresponding representative on the server side, the processing of the EventMessage is initiated. This processing is done within the corresponding instance of the type `PublisherNetInf`. The translation of the `EventMessage` is delegated to an overridden method within a subclass of the afore mentioned class. In a second step, the translated event is published within the instance of the subclass of the type `EventServiceNetInf`. The complete publishing process is defined within the concrete implementation of the **ES**, and does not have to be known within the **ESF**.

The subscriber side works in a similar fashion. The main difference is that two translations have to be implemented, which are the translation of `NetInf`-specific `ESFSubscriptionRequests` to subscriptions of the target **ES**, and the back-translation of the events of the target **ES** to `NetInf`-specific `EventMessages`. Thus, the `SubscriberNetInf` contains two different processing methods, and two different translating methods which have to be implemented for each underpinned **ES** implementation.

Subscriptions are represented by SPARQL [11] queries within `NetInf` which are given within the `ESFSubscriptionRequest`.

3.8.1.2 Functionalities of the ESF

As already mentioned, the **ESF** provides additional functionalities which are automatically added to arbitrary implementations of **ES**s. These different functionalities and their positive effect on `NetInf` are described in the following paragraphs.

Automatic Expiration of Subscriptions Every `ESFSubscriptionRequest` contains a relative expiration time, measured in seconds. This means that the according subscription is active for the given period of time, before the **ESF** automatically removes the according subscription. By that it becomes possible to state interest in the change of `IOs` for a defined period of time. This might be necessary if one is interested, e.g., in the change of the traffic situation for a particular period of time and does not want to manually remove the according subscription. This functionality is realized by the class `SubscriptionExpirationController`.

Local Reliability The local reliability is increased by storing the subscriptions and pieces of information about the subscriber within a database that is running locally to each **ESF**. The `SubscriberDatabaseController` is in charge of keeping a consistent state between the main memory and the database. By that, it is possible to shutdown an event broker, and restart the event broker without losing pieces of information about the subscriber and the associated subscriptions of that subscriber. The according subscriptions are once again sent to the concrete implementation of the **ES**.

This comes in handy when the event broker breaks down unexpectedly. With the help of the `SubscriberDatabaseController` it is always possible to restore the set of issued subscriptions automatically. Thus, the local reliability of the **ES** is increased.

Storage of Events Within the Information-centric Network Subscribers do not have to keep their connection to the **ES** alive all the time. Their subscriptions do not become void, when they disconnect from the **ES** and once the connection to the **ES** is re-established, they may ask the **ES** to deliver all events that have been matched by their subscriptions in the meantime and could not be delivered immediately because the subscriber was disconnected.

Obviously, the **ES** needs some kind of persistent storage for this purpose. One option would be to store those “missed events” in a local database. But this would imply that the subscriber has to connect to exactly the same event broker to which he was connected, in order to receive the missed events. Another possibility is to store them in the **RS** as regular **IOs**. This possibility leverages **NetInf** and provides the functionality of fetching the missed events from any event broker. We decided to use the latter solution. The identifier of those **IOs** is chosen by the **ES** based upon the subscriber’s **IdO** identifier that has been specified in the **ESFRegistrationRequest**.

In order to fetch the events that occurred while the subscriber was offline, he sends an **ESFFetchMissedEventsRequest** to the **ES**. The **ES** then fetches the missed events from the mentioned **IO**, afterwards sends them to the subscriber as part of an **ESFFetchMissedEventsResponse** and, finally, automatically removes them from the **IO**.

A “peek” functionality that would fetch the missed events without removing them from the **IO** could be trivially implemented by means of an additional boolean field in the **ESFFetchMissedEventsRequest**, but our scenarios do not need this functionality. Furthermore, events that are stored in an **IO** because the subscriber is offline are currently stored unencrypted and can therefore be read by anyone. The **ES** should encrypt them to prevent the disclosure of sensitive information.

3.8.2 Event Service Siena

Our **ESF** does not dictate the use of one specific **ES** implementation. Currently, only support for *Siena* has been implemented, but adding support for other **ES** implementations is possible. We have chosen the “Wide-Area Event Notification Service” *Siena*¹³ because it is reasonably stable and has been written in Java, like all of our **NetInf** components.

Additionally, we chose *Siena* because of the suitable routing scheme for subscriptions and events. The routing scheme broadcasts subscriptions and is thus able to route events on the shortest possible path to their subscribers. This is suitable if we have some subscriptions which claim interest in a large amount of events. Exactly this characteristic holds for nodes providing **SSs**. We only have some **SSs**, claiming interest in a large amount of events.

The code that glues the **ES** *Siena* to our **ESF** consists of four major components:

- The translation of **NetInf** Events to Siena Notifications
- The translation of Siena Notifications back to **NetInf** Events

¹³<http://serl.cs.colorado.edu/~serl/dot/siena.html>

- The translation of **NetInf** Subscriptions to Siena Filters
- The startup and connection to other event brokers running *Siena*

Notice that subscriptions respectively filters have to be translated in only one way whereas support for the translation of events respectively notifications has to be implemented in both ways. That is because the input required by an **ES** implementation like *Siena* consists of Filters and Notifications whereas the output only consists of Notifications – there is no need for *Siena* to pass Filters back to the **ESF** and hence support for translating Siena Filters to **NetInf** Subscriptions is not needed.

Translation of NetInf Events to Siena Notifications **NetInf** Events are described by the tuple (old **IO**, new **IO**) where the old **IO** is `null` if an **IO** has been created and the new **IO** is `null` if an **IO** has been deleted. If both elements of the tuple are set then this indicates the change of an existing **IO**. These **NetInf** Events are created whenever an **IO** is created, modified, or deleted. All these changes can be expressed by sending the old and the new state of the **IO** to the **ES**.

Each **IO** consists of an arbitrarily deep hierarchy of attributes. There is no restriction for two attributes to have different **AttributeIdentifications**, which might be seen as some kind of keys for attributes.

In contrast to that, Siena Notifications consist of key-value pairs and within a Notification the keys have to be unique. Hence, if multiple attributes of an **IO** have the same **AttributeIdentification**, then a Siena Notification cannot adequately reflect the **NetInf** Event – we have to arbitrarily choose one of those attributes with the same **AttributeIdentification** and skip the others.

Since **NetInf** Subscriptions respectively Siena Filters can query concrete values of attributes or the change of an attribute value, we have introduced the following prefixes for the keys of Siena Notifications:

old Contains the value of an attribute of the old **IO**

new Contains the value of an attribute of the new **IO**

diff Contains one of the predefined values `CREATED`, `DELETED`, `CHANGED` or `UNCHANGED` depending on the difference of an attribute value between the old and new **IO**

diff_details Contains either “`<`” or “`>`” if a numeric attribute value has changed

Translation of Siena Notifications to NetInf Events Actually, this translation does not really happen in our integration of the *Siena* **ES**. The reason is that, as already described, the translation of a **NetInf** Event (old **IO**, new **IO**) into a Siena Notification is not lossless. It would be impossible to reconstruct the **NetInf** Event from the Siena Notification unless we add additional pieces of information to the Notification. Accordingly, we serialize the **NetInf** Event and append this serialized representation of the original **NetInf** Event as a special attribute.

Then, when *Siena* passes the `Notification` back to the `ESF` because it has been matched by a subscription, the described trick allows us to completely reconstruct the `NetInf` Event by simply deserializing the mentioned special attribute of the *Siena* `Notification`. Here, the trade-off is obviously the increased size of *Siena* `Notifications`.

Translation of NetInf Subscriptions to Siena Filters `NetInf` Subscriptions use a small subset of SPARQL [11]. In particular, GROUP BY, ORDER BY, HAVING, OFFSET and LIMIT are not supported and the result variables have to be `?old` and `?new` (indicating that the old and the new `IO` will be returned).

WHERE clauses may contain triples and filters. The subject of a triple has to be a variable, like for example `?old` or `?new`, the predicate has to be the URI of an `AttributeIdentification` and the object may either be a literal or a variable.

Filters can be used to compare attribute values with other operators than the equality operator. They can additionally be used to check whether an attribute has been created or deleted.

Listing 3.4: Sample subscription query

```
SELECT ?old ?new WHERE {
  ?old <http://rdf.netinf.org/2009/netinf-rdf/1.0/#name> "Donald
    Duck".
  ?new <http://rdf.netinf.org/2009/netinf-rdf/1.0/#name> ?name2 .
  ?old <http://rdf.netinf.org/2009/netinf-rdf/1.0/#person_age> ?
    age1.
  ?new <http://rdf.netinf.org/2009/netinf-rdf/1.0/#person_age> ?
    age2 .
  FILTER (!bound(?name2)) .
  FILTER (?age2 > ?age1) . }
```

Listing 3.4 shows an example for a subscription query. The query matches `IOs` that have been modified as follows: The old `IO` has an attribute with the `AttributeIdentification` *name* whose value is *Donald Duck* that no longer exists in the new `IO` (“`FILTER (!bound(?name2))`”). Furthermore, both `IOs` have an attribute whose `AttributeIdentification` is *person_age* and whose value has increased.

Startup and Connection to Other Event Brokers Running Siena *Siena* can run distributed over many machines. Whether an instance acts as the master depends on the configuration setting `siena.master_broker`. If this property is left unset then this instance will act as a master, otherwise it will connect to the master broker specified by an IP address.

Chapter 4

Generic Path Integration

DEPRECATED — USAGE NOT RECOMMENDED

Note Along with the development of the new OpenNetInf prototype the [GP](#) part was marked as deprecated and should not be used anymore. It is planned to introduce a replacement for [GP](#) in a future version.

Generic Path ([GP](#)) is an approach to normalize and unify the access of functionality of the different layers within the ISO/OSI network stack. Accordingly, because of the unified access, it becomes possible to wire layers in an arbitrary way, without having to adhere to the defined ISO/OSI stack. For a complete description of the [GP](#) approach, please have a look at the separate [GP](#) documentation of our project group.

[NetInf](#) uses [GP](#) in multiple ways. The first and foremost usage of [GP](#) is to find other nodes which provide a particular functionality, like, for example, the possibility to resolve an identifier to an [IO](#) or the search for identifiers, whose [IOs](#) contain certain pieces of information. This usage is described in Section 4.2. Another functionality provided by [GP](#) and used within [NetInf](#) is the possibility to transfer binary data. Such a transfer is equipped with several advantages provided by [GP](#). A description of this kind of usage is described in Section 4.3. But before concentrating on these defined ways of using [GP](#), Section 4.1 introduces the connection between [GP](#) and [NetInf](#).

4.1 Connection to GP

A [NetInf](#) Node might be connected to at most one [GP](#) instance. The typical way to achieve this, is to run a [GP](#) instance on the same host as the [NetInf](#) Node. A node is represented by a node entity within [GP](#)¹. All node entities belong to the same node compartment. Accordingly, each node entity has the possibility to find each other node entity, and can subsequently communicate with one particular node entity.

¹For additional pieces of information about entities and compartments have a look at the separate GP documentation.

4 Generic Path Integration

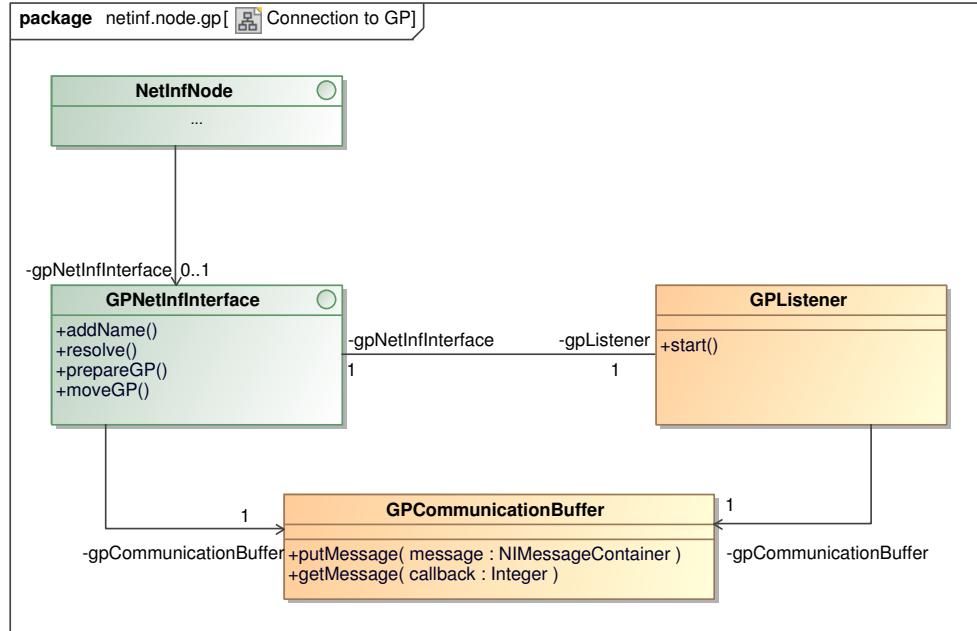


Figure 4.1: Connection to GP

The following descriptions are based on Figure 4.1. A connection to the locally running **GP** instance is represented via an instance of the type **GPNetInfInterface**. Technically, the connection uses a TCP socket, where the **GP** instance maintains a server, and the **NetInf** Node connects to this server on startup. The communication between **NetInf** and **GP** is realized via protobuf messages, where each message is wrapped into a message of the type **NIMessageContainer**.

Each node that is connected to **GP** maintains exactly one instance of the type **GPNetInfInterface**. Method calls to this object result in messages which are directly sent to the local **GP** instance, which then triggers the **GP** functionality described in the following sections. The instance of the type **GPLListener** listens for messages and associates these messages with their requests previously sent by the node. Accordingly, **GP** only sends replies, if **NetInf** has sent a request previously. This is done with the help of the **GPCommunicationBuffer**, which can be seen as a large mail box capable of associating requests and responses.

4.2 Name Resolution

The name resolution addresses the problem of finding nodes in one's neighborhood that are capable of, for example, resolving an identifier to an **IO** or searching the space of **IOs**. The functionality is achieved by two methods defined within **GPNetInfInterface**, which are `addName()` and `resolveName()`. Both methods base on the **GP – NetInf** data model visually depicted in Figure 4.2.

GP defines capabilities for every node. Capabilities and their properties express the

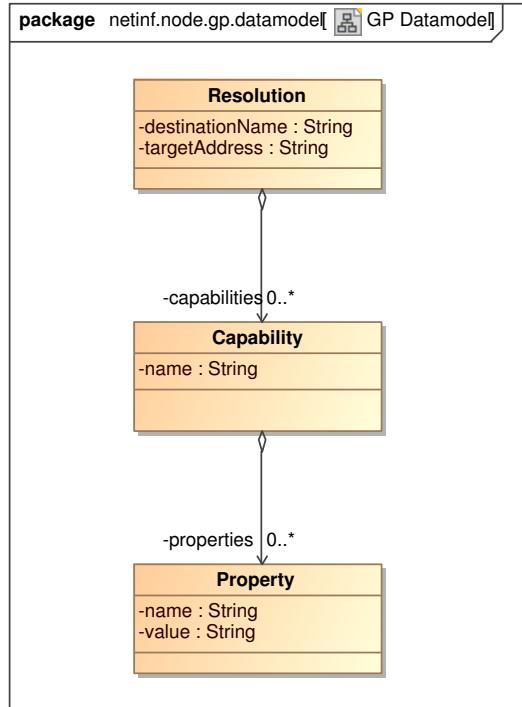


Figure 4.2: Data model for the communication between GP and NetInf

characteristics and functionalities of a node. They define, for example, whether a node can resolve identifiers to `IOs`. Similarly, capabilities can express characteristics of a node like the bandwidth of the connection to the node.

Capabilities and properties are represented by instances of the classes `Capability` and `Property`. Each capability has a unique name. Within our implementation we only use the capability `netinf.common`. The name `netinf.common` simply defines the namespace for the subsequently defined properties – and should not be confused with the package `netinf.common`. A property is a simple name value pair. For expressing the possibility to search, an instance of the type `Property` with the name *function* and the value *search* would belong to the capability with the name `netinf.common`. The name of a property does not have to be unique within a capability. If a node provides several functions, several properties with the name *function* might exist.

Add Name In order to announce the capabilities of a node, each node sends a message at the startup by calling the `addName` method of the `GPNetInfInterface`. This method takes the name of the current node – a string representing the node’s identifier – and a list of capabilities, and sends an appropriate message to `GP`. By that, `GP` knows the capabilities of the according `node` entity and is able to look for this node, whenever a request for a node with these capabilities and properties is issued.

The capabilities and their properties are read from the global configuration file of the node. The functionality is provided by `CapabilityParser`. This might be changed one day, by providing the possibility to infer the required pieces of information au-

tomatically, like the functionality provided by the node, its bandwidth, and its kind. Accordingly, the characteristics of a node might change during lifetime, and would then have to be resent.

Resolve Name In order to find a node with certain characteristics, the method `resolveName()` of the type `GPNetInfInterface` can be used. This might be necessary, if, for example, all other `RS`s within one node were already tried in order to resolve an identifier and, finally, only the `RS RemoteResolution` remains. Thus, `resolveName()` can be used as some kind of fallback, if the query issued to this node could not be processed locally.

For resolving, the node sends a list of required capabilities to its `GP` counterpart, and receives a list of `Resolution` as a result. The required capabilities are determined locally, and might e.g. result from the inability to process a request locally. The capabilities are defined with the help of instances of the class `CapabilityDeterminator`.

A `Resolution` contains pieces of information of the node fulfilling the required capabilities. The name of the node is stored in `destinationName`. Its TCP address is stored in `targetAddress`. The list of `Resolution` instances is handed over to a `ResolutionSelector`, which is then in charge of determining the most appropriate node, in case several possibilities for solving the request exist.

A connection to this other node can then be created with the help of an instance of the type `RemoteNodeConnection` (see Section 3.2.4). The functionality described above for everything concerning the `RS`s is implemented by the class `GPResolutionService`.

4.3 Transfer

An entirely different application of the functionality provided by `GP` is the usage of `GP` to transfer data. This section describes one possible usage of the provided functionality.

The functionality is used in the following scenario: A video-stream is started from the InFox (see Section 5.1.2), which issues a request to the `NetInf` Node. The node forwards this request, which results in opening a video player for showing the particular stream. While showing the stream, the InFox issues a second request to the `NetInf` Node to relocate the video stream to another video player. The forwarded message yields the setup of a new video player, and the shift from the old one to the new one. This scenario shows the functionality of `GP` to relocate end points, without interrupting the data transfer.

The scenario results in some functionality that has to be provided by the `NetInf` Node. It must be able to forward two messages for starting and changing the video stream. The according `NetInfMessages` are called `TCStartTransferRequest` and `TCCChangeTransferRequest`. This is done with the help of the two methods `prepareGP()` and `moveGP()`. The first message takes a source which has to be the URL of the according video stream. The second message takes a string which represents the name of the `targetEntity`. This entity can be seen as a representative of the

second video player instance. It is represented in **GP** by a video entity, which has a unique name.

Chapter 5

OpenNetInf Scenarios and Use Cases

To show the advantages of **NetInf**, we have considered three scenarios in which applications use the **NetInf** infrastructure. The scenarios and some of the applications used within them are described in this chapter.

5.1 Scenario 1: Data Integrity and Data Availability

The first scenario demonstrates two main advantages of **NetInf**: *data integrity* and *data availability*. We developed a Mozilla Firefox extension called InFox, which enables us to use NetInf-URIs within web pages (see more in Section 7.1). A NetInf-URI does not refer to a location or server but to a **DO** (see Section 3.6.1). The **DO** stores several HTTP-URLs of the actual web page in special attributes called locators (`http_url`) and a hash value of the data itself stored in another attribute (`hash_of_data`). The data in this case may be e.g. the HTML file itself. InFox now resolves **NetInf** links to HTTP-URLs via a **NetInf** Node. This node may be a local node or trusted remote node.

Since many **NetInf** Nodes are interconnected, **IOs** can be found no matter on which node they are stored. The node to which InFox connects caches all requested **DOs** and their corresponding **IdOs**. It can also cache the binary data, e.g. the HTML file, which is referenced in the **DO**. Before anything is cached, the integrity is checked. Binary data is checked via its hash value, which is also part of the **DO**. For details on how the integrity of a **BO** is checked, see Section 3.6.

5.1.1 Scenario Description

The scenario contains three parties – see Figure 5.1: **NetInf** Nodes A and B which are running on end-user systems with InFox as the client application. **NetInf** Node C is an infrastructure node running on a server.

At the beginning of the scenario only node C stores the **IO** `IOTEST`. This **IO** dereferences a picture, the data of which is stored on two different servers. One of the servers holds a corrupted version (i.e. a slightly different image).

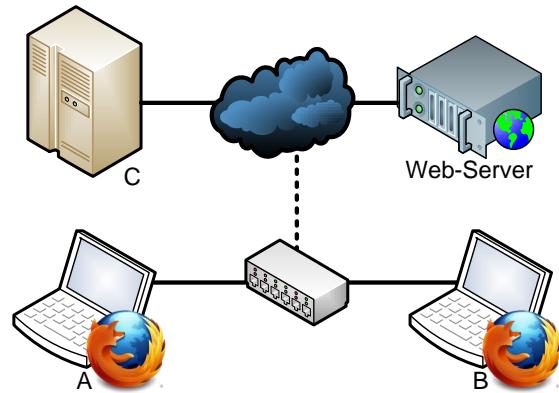


Figure 5.1: Scenario 1: A and B are end-user NetInf Nodes with the InFox extension and C is a infrastructure NetInf Node.

5.1.1.1 Data Integrity

The InFox user at node A requests the picture via a NetInf-URI on the demo page, which can be seen in Figure 5.2. What now happens is that node A searches for other NetInf Nodes and finds node B, an end-user node, and node C, an infrastructure node. Since infrastructure nodes have a higher priority, node A requests the IO, represented by the NetInf-URI, from node C. Node C sends the IO to node A. Node A now looks for the http_url attributes within the IO and starts the download and integrity check of the picture. If it gets the correct picture, it will cache it and deliver it to Firefox. If it gets the corrupted picture, it will try the next http_url location until no more such locators are found.

Augmented Internet 2 Demo

Data Integrity, Data Availability

This InFox is connected to node A



- Correct image
- Incorrect image
- Here's the netinf anchor to the 4WARD Logo

Figure 5.2: InFox demo page

5.1.1.2 Data Availability

The second part of the scenario demonstrates data availability. Node A and B are now getting disconnected from the Internet so that they can neither connect to node C nor to any web server. A and B are still able to communicate with each other. Now the InFox user at node B requests the same picture as in the first part of the scenario. Since node A cached both the **IO** and the picture, it can now deliver these to node B without requiring an Internet connection.

5.1.2 Applications

We have developed two applications which combine the existing Web with the advantages of [NetInf](#). As mentioned above, there is the *InFox*, an extension for Mozilla Firefox¹ which is described in Section 7.1 in detail. Our second application is *InBird*, which is an extension for Mozilla Thunderbird² (see Section 7.2).

To run InFox in this scenario correctly it is recommended to disable the usage of the RESTful interface in the preferences. The old behavior of InFox is expected here.

See Section 6.7 how to install these Mozilla extensions.

5.1.3 How to Run the Scenario

The described scenario has been automated to a great extent. The components can be started automatically. After deploying (see Section 6.1.4), the folder build/scripts/scenario1 contains the scripts needed to start the scenario. Please remember to adjust IP addresses contained here and in the build/configs/scenario1 folder. The addresses are from the 10.10.10.0/8 network which is correct for the demo environment and have to be adjusted if you do not use that address range. The scripts startMachine*.sh automate the process of starting the logging server and the three nodes. The infrastructure is then started on node C, end-user nodes are started on nodes A and B together with a web browser (Iceweasel in this case). The scenario is then ready to be executed.

5.2 Scenario 2: Update Awareness

The second scenario shows three other advantages of [NetInf](#). Using the [ES](#) it is possible to get nearly live updates of data changes without the need for polling. For example, this can be used for collaborative document editing or to keep search indexes up to date. The combination of search feature and [ES](#), furthermore, allows to easily realize specialized global [SSs](#) by subscribing to [IOs](#) with specific characteristics. Finally, the information model ([IdOs](#)) and security concept (authorized reader/writer,...) of [NetInf](#) allow an easy handling of encryption and access rights to information.

To show these advantages we use a shopping scenario which consists of two parts. First, think of the following situation: You live together with another person in one household, e.g. your boy- or girlfriend, and thus you have a common shopping list. If one of you is at home and adds a product to the list and the other one is out in town it would be nice if the latter gets to find out about the added product without calling the other one on his mobile phone.

Second, think of the following: You have a shopping list with some products on it. Now, if you walk around town, you always have to look at your shopping list or remember the products you want to buy and have to consider whether some nearby shop may offer one of these products. It would be nice if some application on your mobile

¹<http://www.mozilla.org/firefox/>

²<http://www.mozilla.org/thunderbird/>

device would do this for you and inform you if you are near a shop which offers a product that you want to buy. It would be even nicer if this application knows whether the product is in stock right now and only informs you in that case.

We realized this scenario with our [NetInf](#) prototype in the following way:

Shopping lists and the inventory of shops are represented as **IOs**. These **IOs** have an attribute which identifies them as being a *ShoppingList* or a *Shop*. They contain all the products that should be bought, respectively are in stock. *ShopIOs* additionally store the geo position of the shop they represent.

The **SS** has a query template which allows to search for shops around a position offering specific products. Furthermore, we implemented two applications which we use to represent the involved persons and shops. More details about these applications are given in Section [5.2.2](#).

5.2.1 Scenario Description

The setup for this scenario consists of several infrastructure and user components as is shown in Figure [5.3](#).

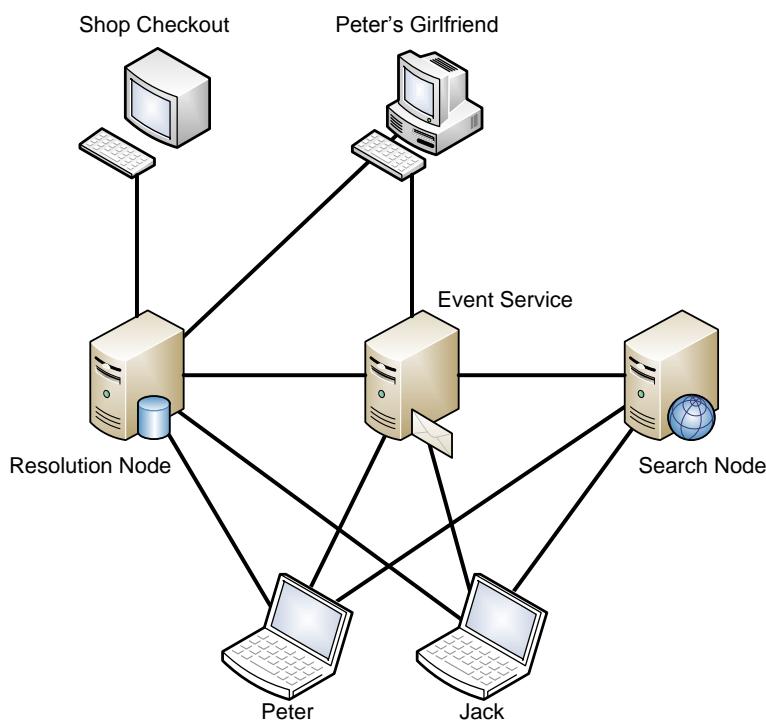


Figure 5.3: Scenario 2: involved parties and their communication

The infrastructure consists of three parts. A resolution node stores all **IOs** relevant for the scenario. It publishes changes of these **IOs** to an **ES**. Furthermore, a **SS** exists, which allows to search for shops around a position which offer specific products. This **SS** is connected to the **ES** and subscribes to changes of **IOs** which represent shops.

A shop, *Walmart*, runs a checkout application which is connected to the resolution node. Two protagonists, *Peter* and *Jack*, both run a shopping application on their mobile device. These applications are connected to all infrastructure components (details about the connections can be found in Section 5.2.2). Furthermore, *Peter's girlfriend* runs an application on her home PC which allows her to manage shopping list IOs.

In the following, the sequence of actions for the scenario is described.

5.2.1.1 Initial Situation

Protagonist *Peter* has an empty shopping list. He starts at his home in Paderborn (*Im Lichtenfelde*). Protagonist *John* has nothing on his shopping list and he is at the technology park in Paderborn. *Walmart* (Suedring) has one bottle of *fresh milk* in its inventory. *Foodland* (Driburger Strasse) also has *fresh milk* in its inventory.

5.2.1.2 Part 1: Protagonist Peter

In the morning *Peter* walks to work at the technology park. Since his shopping list is empty, his mobile application does not search for nearby shops. While he is at work, his *girlfriend* adds *fresh milk* to his shopping list. After work, he walks back home and since he has a product on his shopping list, his mobile application searches for nearby shops offering *fresh milk*. His application notifies him that *Walmart* offers *fresh milk*. He goes there and buys the milk. In addition, he updates his shopping list by removing the *fresh milk* from it. *Walmart* also updates its inventory and since it only had one bottle left, no more milk is available.

5.2.1.3 Part 2: Protagonist Jack

Now *Jack* starts walking towards *Walmart*. He has *fresh milk* on his shopping list, so his mobile application searches for nearby shops which offer it. Since *Walmart* no longer offers *fresh milk*, he is not notified by his application. Without his mobile application, he would have gone into *Walmart* to look for milk. He moves on and after some time he reaches *Foodland*. Now his application informs him that *Foodland* offers *fresh milk* and thus he walks in and buys it.

5.2.2 Applications

To demonstrate this scenario, we developed two special applications: a *ShoppingTool* and a *ProductListTool*. These applications are described in more detail in the following subsections.

5.2.2.1 ShoppingTool

The ShoppingTool is used for the protagonists *Peter* and *Jack*. It represents their mobile applications and allows simulating their movement on a map.

Connection to Infrastructure The ShoppingTool application communicates with different NetInf infrastructure components (see Figure 5.4).

It uses the resolution node to load the *ShoppingListIO* of the current protagonist and to update it if some product is removed from the list because it was bought. Furthermore, the node is used to fetch the *ShopIOs* of the shops found while searching. The application sends search requests to the search node to query for shops near the protagonist offering products that he wishes to buy. Finally, the application is connected to the ES. It subscribes to changes of the currently loaded *ShoppingListIO* to automatically get informed if the girlfriend of the protagonist updates the shopping list.

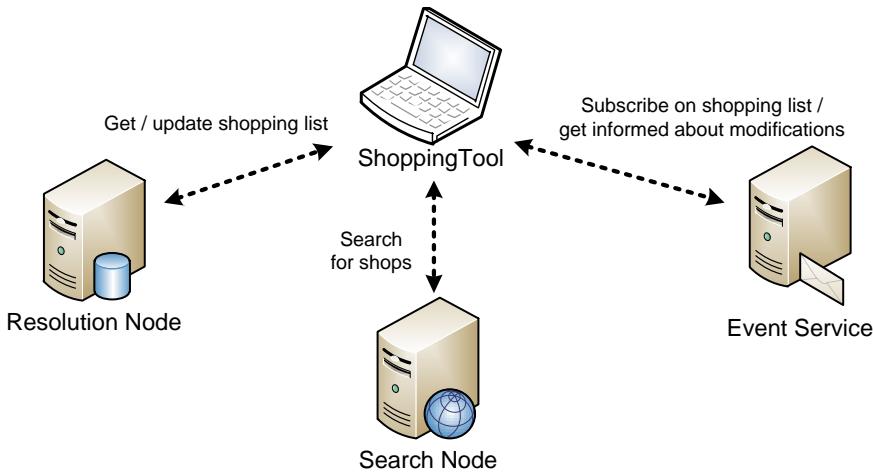


Figure 5.4: Communication of the ShoppingTool with the NetInf infrastructure

Usage This paragraph gives a short overview on the usage of the ShoppingTool. Have a look at Figure 5.5 which shows a screenshot of this tool. It consists of four parts. On the top there is an option panel. Below this, there is a map which can be used to simulate movement in the area. On the right side you see the shopping list of the current user. At the bottom you see the result of the last search.

On the top you can choose the shopping list and thus the protagonist that is represented by the tool. If you choose a list, this list is loaded and the products that are contained in this list are shown on the right. Now the map can be used to simulate movement of the protagonist in the area. His current position is the middle of the blue circle. The circle itself represents the area around his position in which shops are searched. The radius of the search area can be adjusted with the radius field in the top panel. In addition, the six blue waypoints on the map represent shops.

Each time the position is changed, a search request is triggered. The result of the search is printed at the bottom of the application. Furthermore, if shops are found that offer needed products, a pop-up window informs about this fact.

If the protagonist buys a product which is on his shopping list, he has to update the list. This can be done by selecting the according entry in the list and clicking the *remove* button.

5 OpenNetInf Scenarios and Use Cases

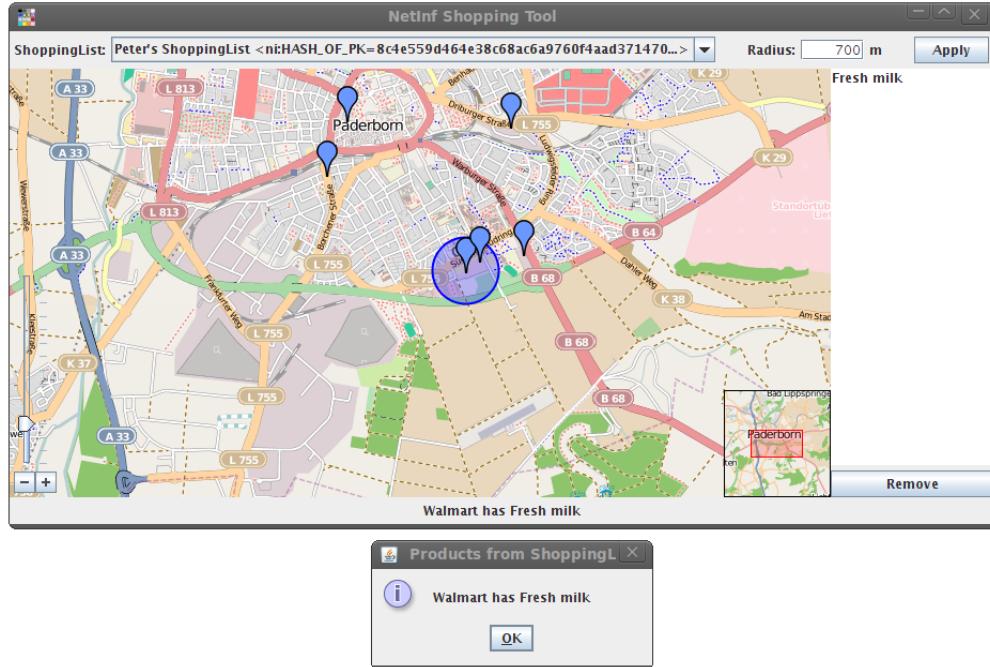


Figure 5.5: Screenshot of the ShoppingTool application

5.2.2.2 ProductListTool

The ProductListTool is used to manage *ShoppingListIOs* of users as well as *ShopIOs* of shops. A *ShopIO* represents an inventory list of the shop containing the products present in the shop. A *ShoppingListIO* is modified by a user. This user might also be a friend or even the fridge of the user owning the *ShoppingListIO*.

Have a look at Figure 5.6 which shows a screenshot of this tool. One can add products to the list, delete products from the list and increase or decrease the quantity of a product on the list. Either a new product list can be created or an existing one can be loaded.

The tool may be tailored to scenario 2 by selecting the parameter for the *application type*: The GIRLFRIEND mode is used to manage *ShoppingListIOs*, the CHECKOUT mode is used to manage *ShopIOs*. In GIRLFRIEND mode the quantity of a product cannot be set and modified.

Furthermore, there is a difference in creating a new list for the two modes as it is shown in Figure 5.7. Whereas the field *Represents* has the value *ShoppingList* in GIRLFRIEND mode, its value is *Shop* in CHECKOUT mode. Additionally, the latitude and longitude of the geo position of the shop have to be set in CHECKOUT mode.

5.2.3 How to Run the Scenario

The described scenario has been automated to a great extent. The components can be started automatically. After deploying (see Section 6.1.4), the folder build/script-

5.2 Scenario 2: Update Awareness

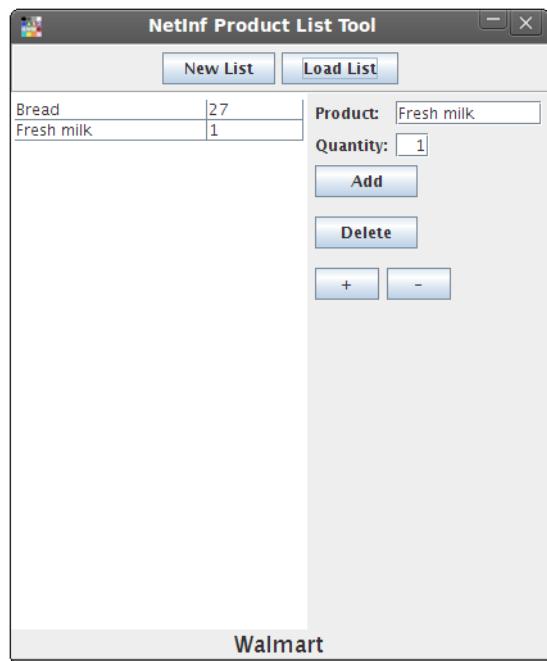


Figure 5.6: Screenshot of the ProductListTool application (not tailored)

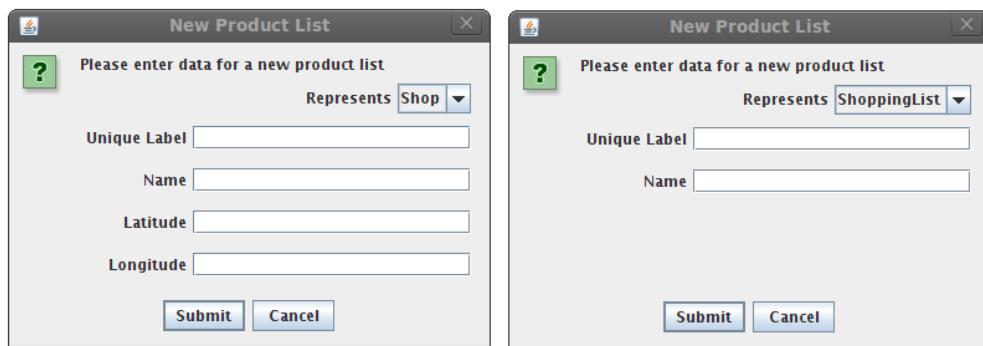


Figure 5.7: Screenshot of the dialog for creating a new inventory list of a shop (left) or a new shopping list (right) in the ProductListTool application

s/scenario2 contains the scripts needed to start the scenario. Please remember to adjust IP addresses contained here and in the build/configs/scenario2 folder. The addresses are from the 10.10.10.0/8 network which is correct for the demo environment and have to be adjusted if you do not use that address range. The scripts start*.sh automate the process of starting the logging server and the involved components. All infrastructure components are running on the same machine and are started via startServer.sh. Two ProductListTools (shop and girlfriend) are started on a second machine via startMachineProductlist.sh and two ShoppingTools (mobile applications) are started on a third machine via startMachineShopping.sh. The scenario is then ready to be executed.

5.3 Scenario 3: Chunking, Streaming, Caching and the MDHT

5.3.1 Scenario Description

The demo basically consists of three **MDHT** levels, four nodes, four clients, two Network Caches – one on each side – and one Peerside Cache. The scenario was developed in order to show the benefits of **NetInf**. There are basically four virtual machines namely *A*, *B*, *C* and *D*, all of which are set up on one physical machine. The complete setup is depicted in Figure 5.8). The first step is to start these four VMs along with the HTTP Server. We have to ping each of the nodes to ensure that they are connected properly. The next step is to start Eclipse (which should be already installed) on each node *A*, *B*, *C*, and *D* and run the launcher in configs/scenario3/runconfigs/<nodeX>. The launching has to be done in this particular order to avoid any issues.

Using the IdentityObject At first, client *A* opens a prepared HTML demopage with **NetInf** URIs using Firefox (the homepage in this case, for ease of use), where the InFox extension is already installed. The outcome is that the links where InFox is already installed, are actually transformed, restyled and formatted as **NetInf** links. The same client clicks on one of the available links (in our demo, the Sintel link) to see where it gets directed to and what can be seen. Then the client clicks on the **IdO** on the same page and it can be noticed that a popup appears which contains the complete information (identity) about the given author, for example *name*, *date-of-birth*, *address*, and *email address*). The latter two are augmented by redirecting to Google Maps and opening an email client, respectively. This clearly portrays the benefits of InFox.

Downloading a Video and Streaming It Client *A* clicks on the Video **IO** which is available on the Sintel demo page. A popup appears with several encodings from which the user may choose from. In this demo the client chooses the ogg-video format. This shows the usage of the REST functionality, since the popup appears near the clicked anchor, which contains information like a *name* and a *description* about the video and also the referenced **DO**, so that client *A* gets to know what kind of video it is and can

5.3 Scenario 3: Chunking, Streaming, Caching and the MDHT

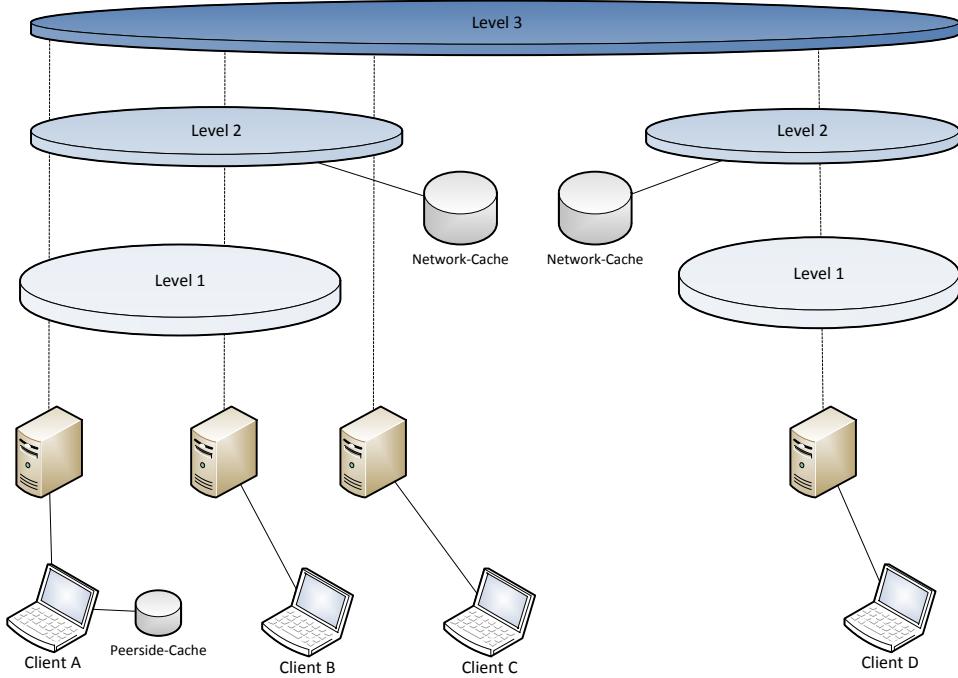


Figure 5.8: Scenario 3 Setup

decide if it has to be downloaded or not after reading the description. A right-clicks the video in order to download it and also decides to play the video. This happens via the Application Interface of the **NetInf** Node and also shows the advantage of the Transfer Dispatcher.

Now, client *B* opens Firefox. It also performs the same action as *A* by clicking on the Sintel link. *B* also clicks on the Video **IO** which opens the popup with the several encodings we have mentioned above. *B* selects the ogg-format, just like *A*, but this time the video is not downloaded to the disk of the client but streamed within the browser. Because the **BO** of the requested video is already cached in the Peerside Cache of client *A*, client *B* receives the video in a chunked and validated way from that source. This demonstrates the feature of Video Streaming which is supported by the Application Interface and the Transfer Dispatcher.

DOs and the Network Cache Client *B* then clicks on the wallpaper **DO** which is available on the browser window. Upon clicking the wallpaper link, the user gets redirected to the image. The regarding **DO** is now cached within the Network Cache.

Later, client *C* opens Firefox, where InFox is already installed. Again the same actions are repeated which includes clicking on the Sintel link and also *C* clicking on the wallpaper **DO**. Since *B* has already clicked on the wallpaper **DO**, it has already been cached and, again, when *C* clicks on it, it will be redirected to the image directly since it actually fetched from the Network Cache.

Publishing new Video Format Client A imports the updated Video IO with a new DIVX format link by using a script, <Node A>/home/netinf/demo/importupdatedio.sh. After this, client C clicks on the Video IO and this time the popup window shows C that a new video format has been added.

MDHT Scope Client B goes to the start page, which is the homepage of the browser. B then clicks on the UPB link and notices that it gets redirected to the HTML page. The same action is performed by client C, that is, C also goes to the start page, which is the homepage of the browser and clicks on the UPB link available on that page. What happens now is that C gets redirected to the error page because the scope of the requested DO is outside client C. The UPB DO has been published up to level 1 and client C is only connected with A and B at level 2 and 3.

This shows the behavior of the MDHT scope and how the visibility of IOs can be restricted to specific domains.

Offline and Different Providers Client D opens Firefox. It also performs the same actions as A and B by clicking the Sintel link.

D also clicks on the wallpaper DO which is then cached on D's provider Network Cache. Client D clicks on the normal HTTP link and it works: That is, it gets redirected to the correct page. And now, the HTTP Server VM can be switched off or paused. When client D clicks on the normal HTTP link now, it can be noticed that it does not work anymore. D clicks on the Image DO and it is still working because the underlying BO has been cached. This is one of the main advantages of NetInf.

5.3.2 How to Run the Scenario

To set up the scenario you need the VirtualBox host virtualization software. Version 4.1.2 was used to create the virtual machines. The VM appliances, called NodeA.ova, NodeB.ova, NodeC.ova and NodeD.ova, as well as DemoServer.ova need to be imported (File, Import Appliance...) in VirtualBox. Complete the wizard. The VM files will be copied on the local computer (the location is configurable in the VirtualBox settings). Before starting the VM, check that there are 2 Network interfaces. The first should be internal, the second should be NAT. This is all there is to it. Now start the VMs, start Eclipse, and in the configs folder you will find the .launcher files in a subfolder scenario3. Launch Node A, then Node B, then Node C, then node D in that order. Afterwards you can follow the scenario description above.

Chapter 6

How to

This chapter offers instructions on how to use the OpenNetInf prototype implementation of [NetInf](#). Section 6.1 explains how to get the sources of the prototype and the required external libraries. Furthermore, it describes how to build the projects and how to run the different components. The customization of a [NetInf](#) Node using *Guice* modules is depicted in Section 6.2. The third *How to* (Section 6.3) is aimed at people who would like to modify parts of our [NetInf](#) prototype or who would like to implement new components such as, for example, a new [RS](#). Section 6.4 describes how to use the [NetInf](#) infrastructure in own applications. Section 6.5 instructs how to communicate with a [NetInf](#) Node using protobuf. Keep in mind that the use of protobuf – as an encoding – is discouraged in the newest version of the prototype. Please use XML instead where possible. In Section 6.6 is the logging functionality explained and finally, Section 6.7 describes the installation of Mozilla extensions.

6.1 How to Get, Compile and Run NetInf Components

The [NetInf](#) sources can be found online. Please visit <http://www.netinf.org> for further details. The sources are split up in several projects which are explained further in Section 2.3.

After you downloaded the sources, there are two main ways to use them. You can either import the projects into Eclipse¹, which allows you to modify, compile and run the [NetInf](#) code in the same environment we used, or you can use *Ant*² to compile a running version of [NetInf](#) as JAR files. Please make sure to have at least *Sun Java JDK 1.6*³ installed on your system.

¹<http://www.eclipse.org/>

²<http://ant.apache.org/>

³<http://java.sun.com/>

6.1.1 Required Libraries

All libraries that are used are listed on the OpenNetInf website⁴. These dependencies are resolved by Ivy what is described in detail in Section 6.1.3.3.

6.1.2 System Requirements

Some **NetInf** components need further preparations depending on your system. The following table lists those requirements.

Components and Requirement	Instructions
RDF ResolutionService, SearchService RDF require: Database (as SDB ⁵ backend)	Configure database as SDB store ⁶ Supported: Derby, H2, HSQLDB, MySQL, PostgreSQL, SQLServer, Oracle, DB2
EventService Siena requires: MySQL	install a MySQL server and insert <code>netinf.eventservice/event_service.sql</code>

6.1.3 Using Eclipse

In this subsection, we assume that you are familiar with Java development in Eclipse. We will not describe the functionalities of the IDE but we will support you in configuring your IDE for **NetInf** development.

6.1.3.1 Preparations

You should at least have the *Eclipse IDE for Java Developers* provided by the Eclipse Packaging Project and the *IvyDE* plugin. To make sure that our code was as good and clean as possible, we used the Eclipse plugin *Checkstyle*⁷ whenever possible. We recommend that you use them for code cleanup as well.

Component	Version we used
Eclipse IDE for Java Developers	3.7.0.v20110530 (Indigo)
Apache IvyDE	2.1.0.201008101807
Checkstyle Plugin	5.3.0.201012121300

Note: Version numbers here are for reference only. They represent a configuration which is known to work. Later versions should be fine, earlier versions should be updated.

⁴<http://www.netinf.org/opennetinf/libraries-licenses/>

⁵<http://openjena.org/wiki/SDB>

⁶http://jenawiki.hpl.hp.com/wiki/SDB/Store_Description

⁷<http://checkstyle.sourceforge.net/>

6.1.3.2 Importing the Projects

You can now use the import wizard (File → Import...) to import *Existing Projects into Workspace* as shown in Figures 6.1 and 6.2.

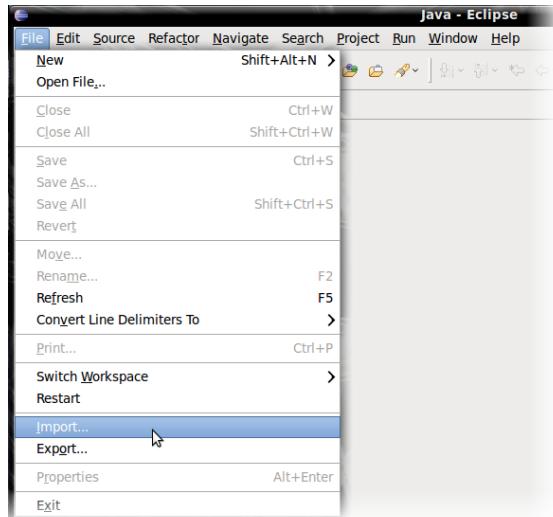


Figure 6.1: Eclipse file import dialog

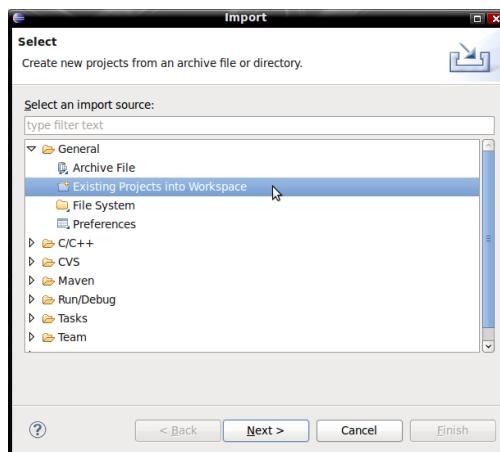


Figure 6.2: Eclipse import wizard

You should select the directory in which the projects (directories named `netinf.*`) reside. Eclipse will automatically select all importable projects. Just click **Finish** here (see Figure 6.3).

6.1.3.3 Ivy

It was mentioned in the “What’s New” section that there is a new way of managing libraries for the **NetInf** prototype. To correctly use Ivy for development you need to

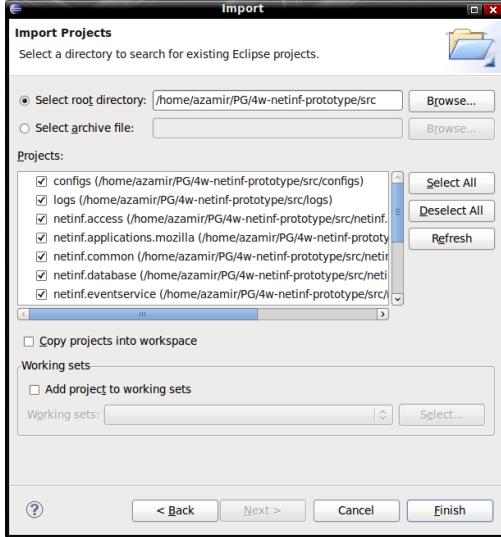


Figure 6.3: Eclipse project selection dialog

install the Eclipse plugin. Go to the IvyDE website⁸ and you will find the detailed instructions on how to install the plugin. The steps are no different from any regular Eclipse plugin, you basically just need the Apache repository URL, which you will find on the website. After installing the plugin you need to restart Eclipse. It is recommended that you clean all current projects. IvyDE will then automatically try to resolve the libraries – found in the ivy.xml file for each project – from the Maven repository.

6.1.3.4 Working with NetInf

You can now modify, compile and run the code directly from within Eclipse. We have included some Run Configurations which you can use to start different components of **NetInf**. The most important Run Configurations are *ES Siena* for the Event Service, *Logging (Standard)* for the Logging Server, *Management Tool Standard Node (Standard)* for the Management Tool and *NetInf Node (Standard)* for the actual node.

6.1.4 Using Ant to Build

After you have checked out the projects you will find a file named `demo.xml` in the `configs` project. This is our main *Ant* build file. To reflect possible changes in libraries, please make sure to check the names of the JAR files mentioned in different parts of the file. However, you do not need to touch this file directly. In order to build a `node.jar`, you simply need to use the `cc-build.xml` file in the root folder of the repository. The command line is then `ant -f cc-build.xml`. This will build the individual projects, then use the `main-build.xml` file with the target `netinf_node` to create the `.jar` file and set the classpath. Ever since version 1.6 the tool is able to set wildcard classpaths, so whatever libraries are used in the subprojects are automatically copied

⁸<http://ant.apache.org/ivyde/>

and used for the `node.jar` file. No modification of the Ant build files is needed. Also, you may use `cc-build.xml` to implement nightly builds, for instance with Cruise Control⁹.

6.1.4.1 Important Build Targets

<code>clean</code>	cleans all .class-files, build-dir and archive
<code>create_archive</code>	creates <code>deploy.tgz</code> in <code>demo</code> folder
<code>deploy</code>	deploy to demo machines (see below)
<code>deploy_local</code>	deploy to local machine, current user

You can use `deploy_local` to deploy a **NetInf** installation to your local home directory. After building the target, you will find a `build` folder there which contains all library JARs, run scripts and **NetInf** JARs as well as copies of the configuration files. The targets are built/executed by going into the `configs`-folder and calling *Ant* with the command line parameters `-f demo.xml target` where `target` has to be replaced by the desired target from the list above (e.g. `ant -f demo.xml deploy_local`).

However, please be aware that as of the newly added Ivy integration, the old build files – with the exception of those described in the section above – may not work anymore or may need to be adapted. Also, you need to install the Ivy Ant¹⁰ plugin, which basically just consists of downloading and adding the `ant.jar` file to the system Java classpath. The `build.xml` in `configs/build` uses this Ant task and will not work without it, as it needs to download the libraries through Ivy.

6.1.5 Command Line Operations

After deploying **NetInf**, you can start the components located in `build/jar` by executing `java -jar jarname.jar`. Some components may require further options to run correctly, please refer to their respective sections for further information.

When the demo was prepared an attempt was made to automate as much of it as possible. As a starting point the scripts are included in the deployment. The scripts might not run on your machine out of the box because they were written for the internal demo machines. They are located in `build/scripts`. Also see Sections 5.1.3 and 5.2.3 regarding those scripts.

6.1.6 Known Problems

Pastry Cannot Bootstrap If you run a node with a Pastry **RS**, make sure to put the correct hostname or IP address in your config file. `localhost` or `127.0.0.1` will not work.

⁹<http://cruisecontrol.sourceforge.net/>

¹⁰<http://ant.apache.org/ivy/history/2.0.0/ant.html>

6.2 How to Create a Customized NetInf Node

We rely extensively on the dependency injection framework *Guice*, as already mentioned. Because of this, it is pretty easy to adjust the provided functionalities of a [NetInf](#) Node. This section gives some brief insights into the actions one has to take to create a [NetInf](#) Node with peculiar functionalities. It assumes familiarity with *Guice* (cp. Section 2.4).

All in all, for each new [NetInf](#) Node a new module, a new subclass of `AbstractModule`, has to be created. To be more precise, a subclass of `AbstractNodeModule` has to be created, which already predefines some standard characteristics peculiar to node modules. Let us call this particular subclass `AdjustedNodeModule`.

The fully qualified class name of this module has to be passed as first parameter to the class starting the whole node. This class is called `StarterNode`. Thus, concerning our example, the node would be started with something like "`java StarterNode netinf.node.module.AdjustedNodeModule`".

If no parameter is passed to the node, the module `netinf.node.module.StandardNodeModule` is used. This is a typical node that contains some exemplary functionality of a node. It is a local node which does not communicate with other nodes. It uses a simple [RS](#), [SS](#) and local cache. Extensive documentation can be found in the according Javadoc.

6.2.1 The AbstractNodeModule Module

Before actually extending the class `AbstractNodeModule` let us have a look on this class itself. It is a subclass of `AbstractModule`, which is the most important *Guice* class for creating separate modules. The most important two parts of the class `AbstractNodeModule` are the constructor and the public void `configure()` method.

The constructor takes one single argument of the type `Properties`. This are standard Java properties, consisting of name-value pairs. These properties are bound to *Guice* such that strings, integer and other "simple types" annotated with the annotation `Named` are automatically injected by *Guice*.

The method `protected void configure()` is automatically invoked by *Guice* during startup. Here e.g. the aforementioned Java properties are bound. Additionally, the standard classes are bound to their implementation in this method. Every [NetInf](#) Node, configured by an extension of the class `AbstractNodeModule` thus has exactly these bindings predefined.

Additionally, the class contains functionalities to configure the connection of the [RS](#) to the [ES](#). The method `provideEventServiceCommunicator`, annotated with the annotations `Provides` and `EventService`, is automatically invoked by *Guice*, whenever a `Communicator` with the annotation `EventService` is required.

6.2.2 The Subclass of AbstractNodeModule

After having analyzed the `AbstractNodeModule`, let us focus on its subclass, e.g. the `AdjustedNodeModule` or the `StandardNodeModule`. Each subclass defines the peculiarities of the according node. First of all, we agreed to define the paths of the properties file within each such subclass. Thus, for example, the class `StandardNodeModule` contains the static final string `NODE_PROPERTIES` denoting the file which belongs to the standard module.

This class has to override the already mentioned `protected void configure()` method. The first statement has to be a call to the super-method. Within this method, we add other modules, which define parts of the node, via the method `install`. By this, all the necessary bindings to start the node, are defined. For example, we define which data model implementation (cp. Section 3.1) is used by the customized node. For other examples have a look at the class `StandardNodeModule` or other subclasses of `AbstractNodeModule` and their corresponding Javadoc files.

Finally, five different methods are used to define the provided services, access methods and interceptors of the node. Each of the methods has to return an array of the corresponding expected type.

Thus, for example, to define the available `RSs`, a method has to be created with the return type `ResolutionService[]`, annotated with the word `Provides`. *Guice* identifies this method as the provider for `RSs` and invokes the method whenever an array of instances of the type `ResolutionService` is expected. Arguments of this method are the `RSs` which should be offered by the node. Notice that we do not create the instances ourselves, but simply advice *Guice* to create the instances, pack them together within this method and return the array of provided `RSs`.

Similarly, the set of `SSs`, `TSss`, resolution service interceptors and access possibilities are defined. For gaining a broader insight, have a look at the Javadoc of the different methods.

6.3 How to Modify, Develop New Components

This section deals with the development of new components. As an example, a new `RS` returning a *HelloWorld* `IO` will be created and explained.

All components used in the `NetInf` implementation are represented through interfaces. Hence, first of all it is important to locate the interface representing the desired component. Most likely it has the name of the corresponding component without whitespaces. Thus, the interface representing the `RS` is `ResolutionService`.

If the goal is to modify an existing implementation it is sufficient to search for all classes implementing the interface and to modify the code of them. The name of classes implementing an interface is usually the name of the interface with a subsequent `Impl`. In cases where more than one class implements an interface this name may differ. The classes implementing the `ResolutionService` interface start with their specialization and end with “`ResolutionService`”: `GPResolutionService`, `LocalResolutionService`, `PastryResolution-`

Service, RemoteResolutionService and RDFResolutionService.

To create a new implementation, a class implementing the interface has to be created. The ResolutionService has four methods to manage IOs: put(), get(), delete() and getAllVersions(). Additionally, it has methods to manage the RS itself: addEventService(), getIdentity() and describe(). For interfaces that are implemented by more than one class, there is often an abstract implementation. The abstract implementation implements methods of the interface that most implementing classes would do in the same way. The AbstractResolutionService implements the addEventService() and getIdentity() methods, but requires a createIdentity() method to be implemented. Additionally, it offers methods to publish changes to the ES.

```
public class HelloWorldResolutionService extends
AbstractResolutionService implements ResolutionService {

    private final DatamodelFactory datamodelFactory;
    private final InformationObject helloWorldIO;

    @Inject
    public HelloWorldResolutionService(DatamodelFactory
        datamodelFactory) {
        this.datamodelFactory = datamodelFactory;
        helloWorldIO = datamodelFactory.createInformationObject()
            ;
        Identifier identifier = datamodelFactory.createIdentifier
            ();
        IdentifierLabel identifierLabel = datamodelFactory.
            createIdentifierLabel();
        identifierLabel.setLabelName(DefinedLabelName
            .UNIQUE_LABEL.getLabelName());
        identifierLabel.setLabelValue("Hello World");
        identifier.addIdentifierLabel(identifierLabel);
        helloWorldIO.setIdentifier(identifier);
    }

    @Override
    protected ResolutionServiceIdentityObject
    createIdentityObject() {
        ResolutionServiceIdentityObject identity = this.
            datamodelFactory
                .createDatamodelObject
                    (ResolutionServiceIdentityObject.class);
        identity.setName("HelloWorldResolutionService");
        identity.setDefaultPriority(1000);
        identity.setDescription("This is a Hello World resolution
            service");
        return identity;
    }
}
```

```

    }

    @Override
    public void delete(Identifier identifier) {
        // There is nothing to delete
    }

    @Override
    public String describe() {
        return "always returning a Hello World IO";
    }

    @Override
    public InformationObject get(Identifier identifier) {
        return helloWorldIO;
    }

    @Override
    public List<Identifier> getAllVersions(Identifier identifier
    ) {
        List<Identifier> list = new ArrayList<Identifier>();
        list.add(helloWorldIO.getIdentifier());
        return list;
    }

    @Override
    public void put(InformationObject informationObject) {
        // This ResolutionService does not support push
    }
}

```

The example **RS** is created as `HelloWorldResolutionService`. It extends the `AbstractResolutionService` and must implement the **RS** interface as explained above. The constructor creates the *HelloWorld IO* the **RS** will return whenever an **IO** is requested. Therefore, it needs a `DatamodelFactory` that is used to create **IOs**. The `@Inject` annotation before methods and constructors is used to indicate to *Guice* which methods have to be used to initialize an instance of a parameter – for example, the aforementioned `DatamodelFactory`.

Since this **RS** is not meant to store **IOs** but to simply return a *HelloWorld IO*, the methods `put()` and `delete()` do nothing. A more sophisticated **RS** – like the `LocalResolutionService` – would save or remove **IOs** from its storage in these methods and publish events to a **ES**. The `get()` method returns the *HelloWorld IO*, no matter which identifier was given for the request. This method usually searches for the **IO** corresponding to the given identifier in the storage (e.g., database or network). The `createIdentity()` method creates the **IdO** required by the `AbstractResolutionService` to identify the **RS** when `getIdentity()` is called. Other **RSs** use this method in the same way as well as the `describe()` method that simply

returns a textual description of the functionality offered by the [RS](#).

To make *Guice* use an implementation, the interface for the component has to be bound to the desired implementing class. This is done in a class inheriting from the `AbstractModule` class *Guice* offers. Either, modules may be installed using the `install()` method. Or interfaces can be bound to classes with the help of the `bind()` method and a subsequent call of the `to()` method on the return value.

```
bind(ResolutionService.class)
    .to(HelloWorldResolutionService.class)
```

If more than one instance should be bound to the interface, a method annotated with `@Provides` is used. This method must return an array containing objects that implement the interface. Its parameters are objects of the different implementing classes supposed to be used. In the method body the objects given as parameters have to be added to an array which is returned afterwards. *Guice* automatically uses this method to inject the required objects into methods having the `@Inject` annotation.

```
@Provides
ResolutionService[] providesResolutionServices(
    LocalResolutionService localRS, HelloWorldResolutionService
    helloRS) {
    return new ResolutionService(localRS, helloRS);
}
```

The binding makes *Guice* use the `HelloWorldResolutionService` class, when one [RS](#) has to be injected. The provided method is used when an array of [RSs](#) is requested. The `LocalResolutionService` and the `HelloWorldResolutionService` are used in this case.

6.4 How to Develop NetInf Enabled Applications

When you develop your own application, you may use [NetInf](#) as a distributed, information-centric storage for data in a machine-readable format. Furthermore, [NetInf](#) provides you with [SSs](#) to search through public data you or others stored in [NetInf](#). Moreover, the [ES](#) is a solution to the problem of regularly polling data, because it lets you register for modification of certain data and notifies you as soon as such a modification occurs. Finally, the integrated security components let you restrict who may read data you stored in [NetInf](#) and who may change it, without setting up individual passwords or access controls.

In the following, this document describes how to use [NetInf](#) in your application. Code fragments are taken from `ShoppingTool` and `ProductListTool` of scenario 2 (see Section [5.2.2](#)). To get details for that code fragments, take a closer look at these tools.

6.4.1 Use NetInf as Data Storage

As the integral part to use [NetInf](#) in your application, you have to include the `net-inf.common` library into it. On the one hand, this library provides you with the [NetInf](#)

data model ([IOs](#), [IdOs](#), attributes, ...) and methods to create [IOs](#) etc. conveniently ([ValidCreator](#)).

On the other hand, the `netinf.common` library provides node communication (`RemoteNodeConnection`) – methods to retrieve existing [IOs](#) from nodes, to modify existing [IOs](#) and to store new [IOs](#) in [NetInf](#).

The [NetInf](#) implementation is built using *Guice*, as described in Section 2.4. When you integrate [NetInf](#) in your application, you may choose to create a new *Guice* module (extending, e.g., the `SecuredApplicationModule`) or you may directly use an existing module (e.g., the `SecuredApplicationModule`). Regardless of whether you create your own module or use an existing one, you should instantiate the module with an individual properties file.

The properties file is used by the `ConfigurableApplicationModule` to decide whether the [RDF](#) or the Java implementation of the data model will be used. Take a look at the properties file for the `ProductListTool` of Scenario 2 (`configs/scenario2/productlist.properties`). `format = RDF` instructs the `netinf.common` library to use the [RDF](#) implementation of the data model. Moreover, the properties file is used to store IP addresses and port numbers of the node to connect to (`cc.tcp.*`) – that is, where you will get [IOs](#) from.

Once you created a properties file and decided which *Guice* module to use, you may actually start implementing:

1. Instantiate Module, e.g.:

```
Module module = new SecuredApplicationModule("path/to/
properties.file");
```

2. Create *Guice* Injector, e.g.:

```
Injector injector = Guice.createInjector(module);
```

3. Instantiate your class by *Guice*, injecting a `DatamodelFactory`, a `RemoteNodeConnection`, and `host` and `port` for that `RemoteNodeConnection`.

E.g., your code looks like the following:

```
public class yourClass {
    @Inject
    public yourClass(ValidCreator vc, DatamodelFactory dmF,
        RemoteNodeConnection rNC, @Named("cc.tcp.host") String
        host, @Named("cc.tcp.port") String port)
    {
        rNC.setHostAndPort(host, port);
    }
}
```

Then you may instantiate that class by

```
injector.getInstance(yourClass.class);
```

6.4.1.1 Create InformationObjects

Given the code above, you may create your own **IOs** containing your data. The **Validator** provides you with methods to conveniently build a new identity and create **IOs** owned by that identity. Their use is straightforward.

6.4.1.2 Retrieve and Put InformationObjects

In **NetInf**, the **IOs** are stored at the nodes. Thus, we use the **RemoteNodeConnection** to retrieve **IOs** and store them.

Assume you have got the string represented identifier of an **IO** you want to retrieve: *someIdentifier*. To retrieve this **IO**, you simply have do to the following:

```
// get Identifier  
Identifier id = dmF.createIdentifierFromString("someIdentifier"  
    );  
// retrieve IO  
InformationObject informationObject = rNC.get(id);
```

Let's say you modified some of the attributes of the **IO** you just retrieved. Now it is equally simple to store the modified **IO** to the node again:

```
io = someModifyingOperation(io);  
// put IO  
rNC.put(io);
```

Using these basic operations, you are able to create, store or retrieve your own or existing **IOs**.

6.4.2 Use NetInf Search

NetInf is designed as a distributed data storage just like the Internet. Thus, **NetInf** very likely contains **IOs** to a wide area of information. To be able to access these **IOs**, you may search within **NetInf** using **SSs**.

Searching for **IOs** works similar to using a remote node for getting **IOs**. First, you establish a connection to a node running a **SS**.

```
public class yourClass {  
    @Inject  
    public yourClass(RemoteNodeConnection searchNodeConnection,  
        @Named("search.tcp.host") String host, @Named("search.tcp  
        .port") String port)  
    {  
        searchNodeConnection.setHostAndPort(host, port);  
    }  
}
```

Once you established this connection, you may send a search query formulated in **SPARQL** [11] to that node. Have a look at Listing 3.2 in Section 3.5.2 and its description for an example. The node will return a list of identifiers of **IOs** that match your query.

```
List<Identifier> identifiers = searchNodeConnection.  
    performSearch("?id netinf:represents ?blank . ?blank netinf:  
        attributeValue 'String:Shop'.", 5000);
```

This query is used in scenario 2 to look for all **IOs** that have an attribute with the identification *netinf:represents* and the value *String:Shop*. Additionally, the *5000* represents the timeout in milliseconds, after which the search request is canceled.

6.4.3 Get Informed by Event Service

As a solution to the problem of repeatedly polling data to get the most current version, **NetInf** provides **ESs**. You may register at an **ES** for creation, modification or deletion of **IOs** matching certain queries. The **ES** will then notify you, if such a creation, modification or deletion occurs.

First, you need to implement the connection to the **ES** and how incoming events are handled.

- 1. Implement an **ESFConnector** by extending the abstract class **AbstractEsfConnector****

```
public class myEsfConnector extends AbstractEsfConnector {

    @Inject
    public myEsfConnector(DatamodelFactory dmFactory,
                          MessageReceiver receiveHandler,
                          AbstractMessageProcessor procHandler, @Named("esf.host") String host, @Named("esf.port") String port) {
        super(dmFactory, receiveHandler, procHandler, host, port);
    }

    @Override
    protected boolean systemReadyToHandleReceivedMessage() {
        return true;
    }
}
```

This code creates a simple **EsfConnector**. **systemReadyToHandleReceivedMessage()** is used to indicate whether or not the **EsfConnector** is ready to accept and handle incoming messages. In case the processing of incoming messages has certain prerequisites, for example, a certain kind of storage, this method may avoid the **EsfConnector** from already processing messages, before the prerequisites are fulfilled. Since we want to (and are able to) handle all incoming messages, no matter when they arrive, we simply return **true** all the time. However, this code does not yet define how incoming messages are handled. This is done by some **AbstractMessageProcessor procHandler** that still has to be defined.

- 2. Implement a **MessageProcessor** extending the abstract class **AbstractMessageProcessor****

If the following **MessageProcessor** is used, for each incoming event, the old (before modification) and the new (after modification) **IO** is printed. In case the incoming event is related to the creation or deletion of an **IO**, **getOldInformationObject()** or **getNewInformationObject()** will return **null**.

```
public class myMessageProcessor extends
    AbstractMessageProcessor {

    @Override
    protected void handleESFEventMessage(ESFEventMessage
        eventMessage) {
        sysout.println(eventMessage.getOldInformationObject());
        sysout.println(eventMessage.getNewInformationObject());
    }
}
```

3. Bind implemented MessageProcessor

Finally, *Guice* has to be instructed to use the implemented MessageProcessor (myMessageProcessor) wherever an AbstractMessageProcessor shall be injected. Thus, you have to modify the *Guice* module you use.

```
bind(AbstractMessageProcessor.class)
    .to(myMessageProcessor.class);
```

4. Get EsfConnector injected

Implementing the previous three steps, you may simply instruct *Guice* to inject a myEsfConnector to your code.

```
public class MyClass {

    myEsfConnector esfConnector;

    @Inject
    public MyClass(myEsfConnector esfConnector) {
        this.esfConnector = esfConnector;
    }
}
```

By the myEsfConnector, you get an EsfConnector that handles incoming event messages in the way you implemented (Step 2). However, you did not register for events yet.

5. Set Identity

A prerequisite to register for events is that you first have to set the identifier of your **IdO idO**, and start the EsfConnector. This identifier is used by the **ES** later on, to determine who some event message is meant for.

```
esfConnector.setIdentityIdentifier( idO.getIdentity() );
esfConnector.start();
```

6. Register for Events

Finally, you may register for events of certain characteristics in IOs. As soon as such characteristics occur in an IO, the MessageProcessor will be triggered. You formulate the characteristics of IOs, you are interested in, by a SPARQL query.

```
esfConnector.sendSubscription("unique Subscription name", "
    SELECT ?old ?new WHERE {?new <http://rdf.netinf.org
        /2009/netinf-rdf/1.0/#identifier> 'someIdentifier'.}",
    600);
```

This line subscribes you for events in all IOs that match the given SPARQL query: All the IOs that have the identifier *someIdentifier* (in fact, this is only one). The subscription is valid for 600 seconds. If you do not want to be notified on events before the 600 seconds elapsed, you may unsubscribe using `esfConnector.sendUnsubscription(...)` with the *unique Subscription name* as its parameter. See Section 3.8.2 for more details on how to formulate a subscription using SPARQL.

6.5 How to Send and Receive Messages Using Protobuf

It is impossible to use the convenient communication methods described in Section 3.2 if one wants to write a NetInf Application in a different programming language than Java. Instead, one has to use raw XML or protobuf messages. The latter approach is the subject of this section.

The following example is implemented in Java but it can be adopted to any programming language that is supported by protobuf and that supports the Transmission Control Protocol (TCP). It sends a ProtoRSGetRequest with the identifier specified in the class constant IDENTIFIER to a NetInf Node and prints out the resulting ProtoRS.GetResponse – most likely an error message since the requested IO does not exist.

```
package netinf.tools.demo;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;

import com.google.protobuf.ByteString;

import netinf.common.communication.AtomicMessage;
import netinf.common.communication.Connection;
import netinf.common.communication.MessageEncoderProtobuf;
import netinf.common.communication.TCPConnection;
import netinf.common.communication.protobuf.ProtobufMessages.
    ProtoNetInfMessage;
import netinf.common.communication.protobuf.ProtobufMessages.
    ProtoRSGetRequest;
```

```

import netinf.common.communication.protobuf.ProtobufMessages.
    ProtoSerializeFormat;
import netinf.common.datamodel.Identifier;
import netinf.common.datamodel.rdf.DatamodelFactoryRdf;

public class ProtobufTest {
    public static final String serverHost = "localhost";
    public static final int serverPort = 5000;
    public static final String IDENTIFIER = "ni:name=value";

    public static void main(String[] args) throws Exception {
        Connection connection = setupConnection();

        ProtoNetInfMessage request = buildRequestMessage();
        System.out.println("Sending request: " + request);
        connection.send(new AtomicMessage(
            MessageEncoderProtobuf.ENCODER_ID, request.
            toByteArray()));

        AtomicMessage atomicMessage = connection.receive();
        ProtoNetInfMessage reply = ProtoNetInfMessage.parseFrom
            (atomicMessage.getPayload());
        System.out.println("Received reply: " + reply);
    }

    private static ProtoNetInfMessage buildRequestMessage() {
        // Build ProtoRSGetRequest
        byte[] serializedIdentifier = serializeIdentifier(
            IDENTIFIER);
        ProtoRSGetRequest.Builder getRequestBuilder =
            ProtoRSGetRequest.newBuilder();
        getRequestBuilder.setIdentifier(ByteString.copyFrom(
            serializedIdentifier));

        // Wrap ProtoRSGetRequest into a ProtoNetInfMessage
        ProtoNetInfMessage.Builder netInfMessageBuilder =
            ProtoNetInfMessage.newBuilder();
        netInfMessageBuilder.setSerializeFormat(
            ProtoSerializeFormat.RDF);
        netInfMessageBuilder.setRSGetRequest(getRequestBuilder.
            build());

        return netInfMessageBuilder.build();
    }
}

```

```

private static byte[] serializeIdentifier(String
    identifierStr) {
    DatamodelFactoryRdf rdf = new DatamodelFactoryRdf();
    Identifier identifier = rdf.createIdentifier();
    identifier.initFromString(identifierStr);
    return identifier.serializeToBytes();
}

private static Connection setupConnection() throws
    IOException
{
    Socket socket = new Socket();
    socket.bind(null);
    socket.connect(new InetSocketAddress(serverHost,
        serverPort), 1000);
    return new TCPConnection(socket);
}
}

```

The `.proto` file in `netinf.common.communication.protobuf` that contains all protobuf messages has to be compiled before it can be used by a programming language. After having installed protobuf, the command `protoc <path-to-proto-file>` generates code in the respective programming language that afterwards can be included or imported. This process depends on the programming language you want to use: have a look at the protobuf documentation¹¹ for details. But fear not, the method names of the generated code are identical or at least similar for each programming language.

The interesting things happen in the method `buildRequestMessage`. There, a `ProtoRSGetRequest` is built and then wrapped in a `ProtoNetInfMessage`. This is a pattern used throughout `NetInf` – all requests and replies have to be wrapped in a `ProtoNetInfMessage`. Communication with a `NetInf` Node always follows a simple request-reply pattern. There is no documentation available in the `.proto` file, but the equivalent Java classes in the package `netinf.common.messages` are equipped with detailed Javadoc comments.

After the request has been built as described it has to be sent over the wire. To let the `NetInf` Node know that we have encoded our request using Protobuf (and not XML), we have to prefix the payload with the respective “encoder id” – in this case 1 (class constant `MessageEncoderProtobuf.ENCODER_ID`). This (and not more) is done by using the `AtomicMessage` class – it allows the class `TCPConnection` to send the “encoder id” concatenated with the payload over the wire. This obviously has to be reimplemented when using another programming language.

Notice that any kind of error handling has been omitted from the example to improve readability. Furthermore, checking the integrity of the received IO is beyond the scope of this section.

¹¹<http://code.google.com/p/protobuf/>

6.6 Logging

We have a distributed system with several applications (see Section 2.1). To be able do debug reasonably, we gather all the logging events at one centralized application which logs all the logging events in a combined fashion. Thus, we can register the present situation at a glance. We use a conversion pattern which contains pieces of information about the application and the hostname, respectively IP:port pair, which send the according request.

We use *Log4j*¹² and decided on the following logging conventions. Here, every level that is of specific importance is mentioned and contains pieces of information about the kind of requests that belong to the according level.

DEBUG The DEBUG level is used to indicate detailed informational events that are useful to debug the applications.

DEMO Everything that belongs to this level is used for presentations. Starting or stopping applications, for example nodes, the IOManagementTool or an ES, belong to the DEMO level.

Messages also belong to this level. Every time a message is sent to another peer, the following line has to be used: `LOG.info("NetInfMessage to be send: \n" + message);`. Every time a message is received from another peer the following line has to be used: `LOG.info("NetInfMessage received: \n" + message);`.

Requests of important interfaces are logged in this level as well. Important interfaces are ResolutionService (get, getAllVersions, put, delete), SearchService (search), Cryptography (encrypt, decrypt), IdentityVerification (isOwnerVerified, isWriterVerified, isIOVerifiedByOwner, isIdentityTrusted) and Integrity (isSignatureValid).

WARN With the WARN level potentially harmful situations shall be marked.

ERROR Only real errors, which imply a problem of running a node correctly, belong to this level. In this sense a failed integrity check or a broken connection between two NetInf Nodes are *no* errors.

The concrete layout looks as follows:

```
X{application}(-15X{host}): %6r %-5p [%t] %c#%M - %m%n
```

- `%X{application}` has to be four letters long and one of the following: esfs for EventServiceFramework Siena, logs for the logging server itself, node for the node, iomt for the IOManagementTool.

¹²<http://logging.apache.org/log4j/>

- (%-15X{host}) is determined automatically. It might either be the name and port (e.g. localhost.localdomain:1234) or the IP and port (e.g. 127.0.0.1:1234).
- %6r is the number of milliseconds elapsed since the start of the program.
- %-5p is the level of the log statement.
- [%t] is the thread making the log request.
- %c#%M is the name of the logger associated with the log request.
- The text after the '-' is the message of the statement.

Each application which wants to send the logging events to our logging server has to have a `SocketAppender`: the `log4j.appenders.SOCK.Application=xxxx` where xxxx defines the application (esfs, logs and so on).

6.7 How to install Mozilla Extensions

There are two different ways to install the InFox and InBird extensions. The first way is to zip the whole InFox, respectively InBird, directory and change the file extension from zip to xpi. This allows the user to drag and drop the xpi file into a Firefox, respectively Thunderbird, window to easily install the extension. The second way is to create a proxy-file in the applications extension folder (e.g. `/.mozilla/firefox/profiles/abcdef.default/extensions/`). You have to create a file called `infox@upb.de`, respectively `inbird@upb.de`, containing the absolute path to the corresponding folder in the repository (e.g. `~/4w-netinf-prototype/src/netinf.applications.mozilla/infox`). Using the second way makes it possible to always use the most recent version of the extension. For further information on how to develop Mozilla extensions please refer to the Mozilla Development Center¹³.

¹³<http://developer.mozilla.org>

Chapter 7

OpenNetInf Applications

7.1 InFox: OpenNetInf Extension for Mozilla Firefox

InFox is an extension for Mozilla Firefox which uses OpenNetInf as a platform for secure and highly available web content. Instead of using regular HTTP links, one may link to an **IO**, **DO** or **IdO** via its Identifier-URI. This brings **NetInf**'s advantages to the today's Web already.

A **NetInf** link, or **NetInf**-URI, consists of the protocol identification `ni:` and an identifier of an **IO**. InFox resolves those **NetInf**-URIs by communicating with the connected **NetInf** Node. Related to the underlying type of the **IO**, InFox handles it in a different way. See below how which type of **IO** is processed.

Note Since the OpenNetInf prototype is enhanced by the Application Interface (see 3.3) InFox communicates not via **NetInf** Messages anymore, but uses the intended RESTful interface to resolve **IOs**. This can be changed to the old behavior in the plugin preferences if necessary (e.g. for Scenario 1).

7.1.1 IdentityObjects

IdentityObjects are representing owners of **IOs** or even components of the OpenNetInf architecture. Therefore it is kind of a business card and contains individual information of that individual. InFox reads that information and displays it within a popup like shown in Figure 7.1.

Thereby InFox parses only the following attributes in addition to the `name` and `description` of the **IdO**:

- `e_mail_address` (also providing a mailto-link to that E-Mail address)
- `person_phone`
- `person_address` (also providing a Google Maps link to that address)
- `person_homepage` (also providing a link to that homepage)

- person_birthday

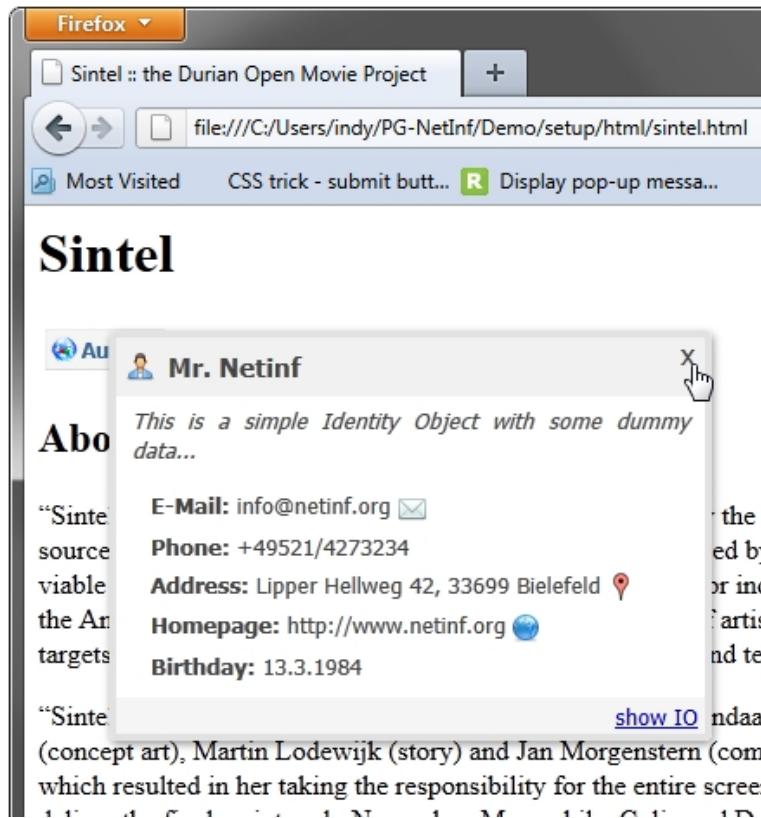


Figure 7.1: Clicking on a IdentityObject

The popup can be closed by clicking outside the popup or by clicking the “X” in the top right corner. If the user likes to see the underlying **IdO** in its XML format he can click on the link “Show IO” in the bottom right corner.

7.1.2 DataObjects

DataObjects contains links to concrete **BOs** of the same type and provides additional information like the hash of the data. If the user clicks on a **NetInf**-URI that represents a **DO** the InFox extension resolves it and redirects the user to the **BO** which is determined by the connected **NetInf** Node – no extra popup is displayed like at **IdOs** or **IOs**.

In Figure 7.2 is demonstrated how a user clicks on a **DO** link within a Video **IO** popup. He is redirected to the related video file and is able to see the video stream inside the browser.

7.1.3 InformationObjects

The **InformationObject** is the most general type to provide information within **NetInf**. If a **IO** contains references to other **DOs**, which can be specified by the attribute `referenced_do`, InFox displays that **IO** within a popup and lists the related **DOs** at the

7.1 InFox: OpenNetInf Extension for Mozilla Firefox

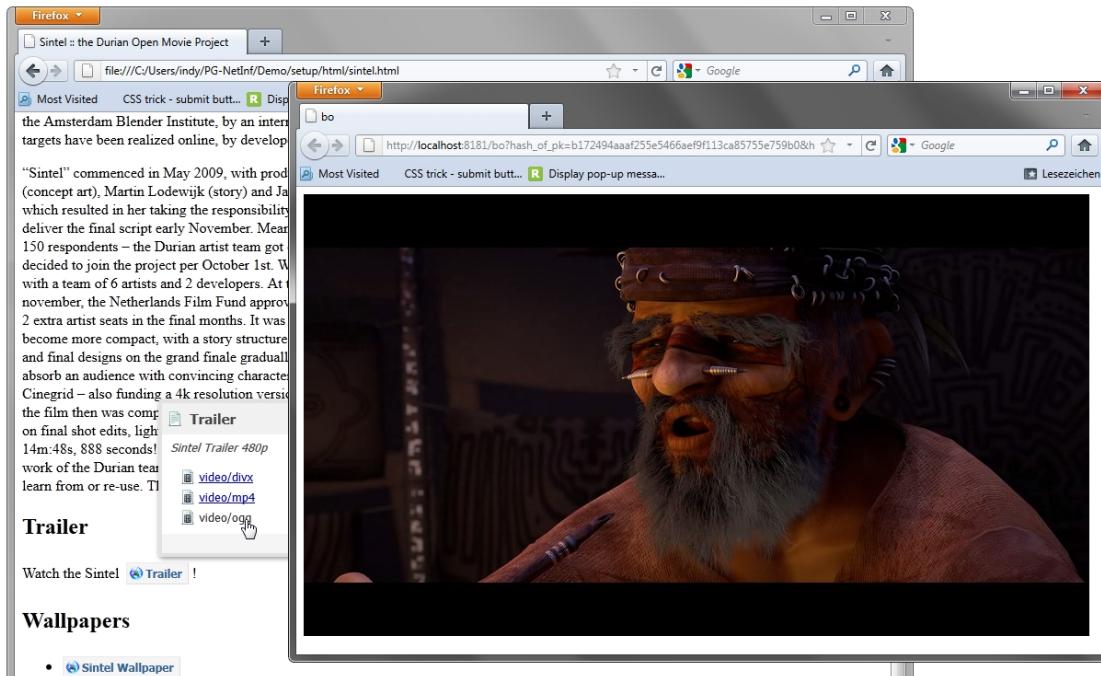


Figure 7.2: Clicking on a DataObject

bottom. If the content type, which can be specified by the attribute `content_type`, is defined as well InFox also reads that information. Additionally the name and description of the `IO` is displayed.

The example in Figure 7.3 shows a Video `IO` that contains several types of the representing Video (Divx, MP4 and OGG). The user can click on one of them and will be redirected as described in Section 7.1.2.

Below you find a snippet of the Video `IO` that shows how a `DO` can be referenced in detail:

```
<netinf:referenced_do rdf:parseType="Resource">
    <netinf:content_type rdf:parseType="Resource">
        <netinf:attributePurpose>USER_ATTRIBUTE</netinf:
            attributePurpose>
        <netinf:attributeValue>String:video/mp4</netinf:
            attributeValue>
    </netinf:content_type>
    <netinf:attributePurpose>USER_ATTRIBUTE</netinf:
        attributePurpose>
    <netinf:attributeValue> String:ni:HASH_OF_PK=b1724...755
        e759b0~HASH_OF_PK_IDENT=SHA1~VERSION_KIND=UNVERSIONED~
        UNIQUE_LABEL=VideoDOMP 4
    </netinf:attributeValue>
</netinf:referenced_do>
```

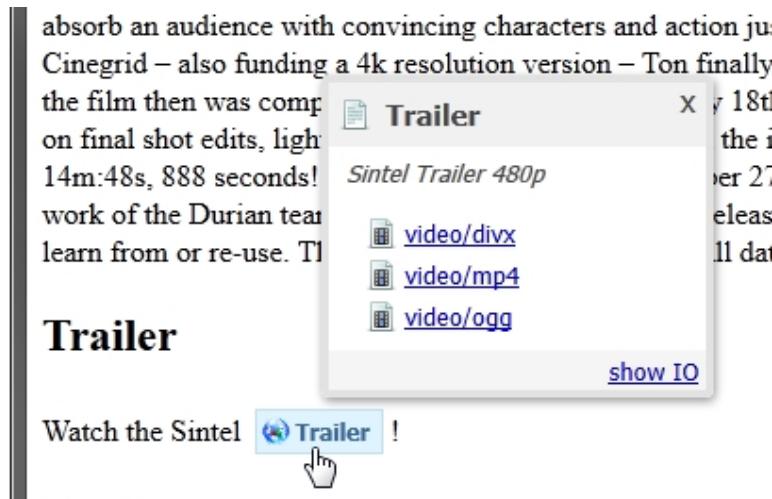


Figure 7.3: Clicking on an InformationObject

7.1.4 InFox Extension Preferences

InFox offers a preferences window like shown in Figure 7.4 to set up the connection to the necessary **NetInf** Node that should be used for resolving **IOs**. The link style can be adjusted to the user's likes and it can be defined if **DO** links should be followed directly or at first resolved and then followed on a second click ("Automatically follow links"). If the "RESTful Behavior with Popups" box is unchecked InFox uses not the Application Interface but the Node Interface with a HTTP interface (old behavior – not recommended!).

A Reset button at the bottom allows the user to reset all preferences to their default setup.

7.1.5 Developing InFox

In order to develop a Mozilla Firefox Extension in an appropriate way, it is recommended to setup at first a productive development environment. Therefore it will be not necessary to generate a xpi-package each time and the extension can be loaded directly from the source folder. Below you will find a set of hints regarding the development of an extension – including how to configure the environment.

- Documentation and API of Firefox Extensions¹
- Setting up extension development environment²
- Javascript online validation tool, JSLint³

¹<http://developer.mozilla.org/de/docs>

²http://developer.mozilla.org/en/setting_up_extension_development_environment

³http://www.javascriptlint.com/online_lint.php

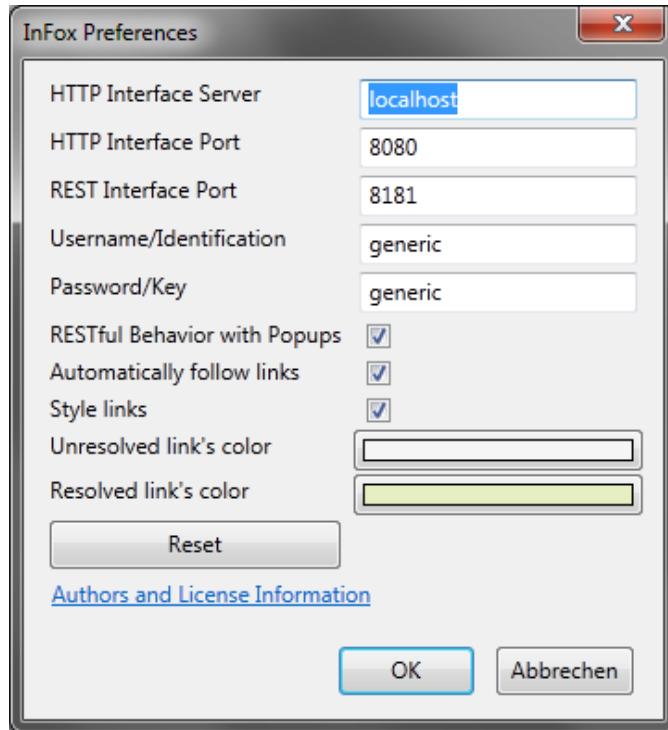


Figure 7.4: InFox Preferences Window

- Javascript extension for eclipse, JSEclipse⁴
- jQuery-API⁵

To improve the development of the extension logging has been used within the code. Thus, you can see what InFox does via Firefox' integrated error console, see Figure 7.5.

7.1.6 How to Generate a XPI-Package

Firefox Extensions are delivered in a so called xpi-packages, i.e. a file called <extensions-name>.xpi. This packages are nothing else than normal ZIP archives that are renamed to XPI. Thus you can simply create such a package by using a tool like 7-Zip⁶ and creating an archive out of the extension folder. Note here is that you should not zip a folder, which contains all the extension files, but create the “folder” by the tool with the file “install.rdf” in top-level for instance. Afterwards the extension can be installed in the usual way like described in Section 6.7.

⁴<http://www.interaktonline.com/products/eclipse/jseclipse/overview/>

⁵<http://api.jquery.com/>

⁶<http://www.7-zip.org/>

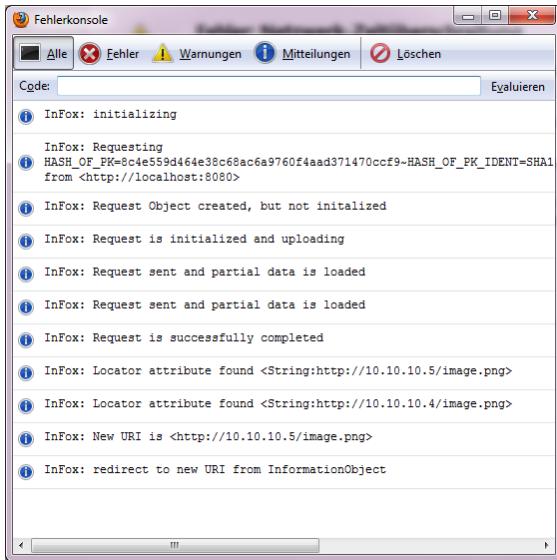


Figure 7.5: Firefox error console showing the InFox log

7.2 InBird: OpenNetInf Extension for Mozilla Thunderbird

InBird is an extension for Mozilla Thunderbird that allows to use **IdOs** as mail recipient addresses. The recipient address has to be in the format `attribute-name@IdO-Identifier`, e.g., `e_mail_address@ni:HASH_OF_PK=8c4e559d4...`.

7.3 Management Tool

To be able to manage and operate our **NetInf** prototype, we implemented the *IOManagementTool*. It can be used to, for example, create, delete, or modify **IOs**. This management tool is discussed in Section 7.3. Furthermore, Section 6.6 describes the centralized logging concept for our prototype implementation.

During the development of **NetInf**, we worked on the infrastructure for a long time without being able to check the results as a whole. Testing parts of the infrastructure is not the same as testing the actual prototype. We also needed a convenient way to display and modify data stored in the prototype. So we developed a simple application which can interact with **NetInf** Nodes. With this *IOManagementTool* it is possible to talk to the Resolution component (display, modify, create, delete, import or export **IOs**) and the Search component (search **IOs**).

The tool keeps a local database of known identifiers. The tool can be started with command line options to define which node to connect to. More options are documented in the source code of its starter class or can be displayed using the `-h` switch. The main window looks as shown in Figure 7.6. From here, the user can choose different actions to perform on the node the tool is connected to. The actual connection to the node is

established and closed on demand.

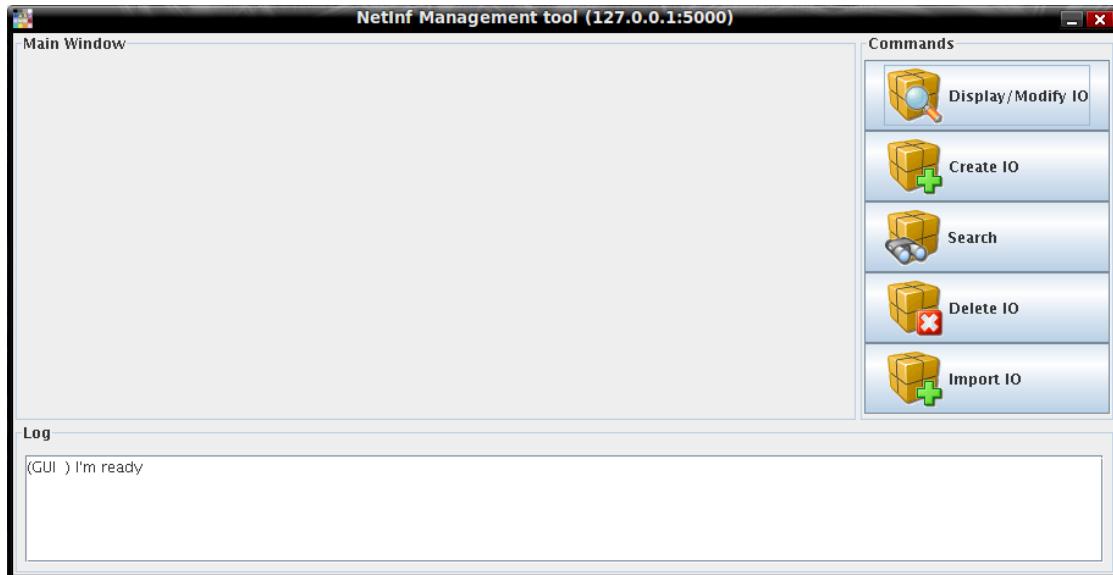


Figure 7.6: IOManagementTool main window

If the user chooses to display or modify an **IO**, he has to enter an identifier. He is then presented with the screen shown in [7.7](#). The **IO** is displayed as a tree on the left while the buttons on the right side change to reflect possible operations on the selected element in the tree. It is possible to add, delete or modify identifier labels, attributes and subattributes.

When adding or modifying parts of the **IO**, the user is presented with the screen shown in [7.8](#). There is support for content assistants (currently only for attribute values which are represented as special strings). They are available through the buttons with three dots. The entered data is automatically checked for validity where applicable. In our example, a valid entry for the attribute identification and an invalid entry for the attribute value have been entered (The string is not an actual Integer representation). The result of the check is shown on the right side (ERR for error, OK, or ??? if unknown). Where applicable, only predefined values are allowed via drop-down menus.

7 OpenNetInf Applications

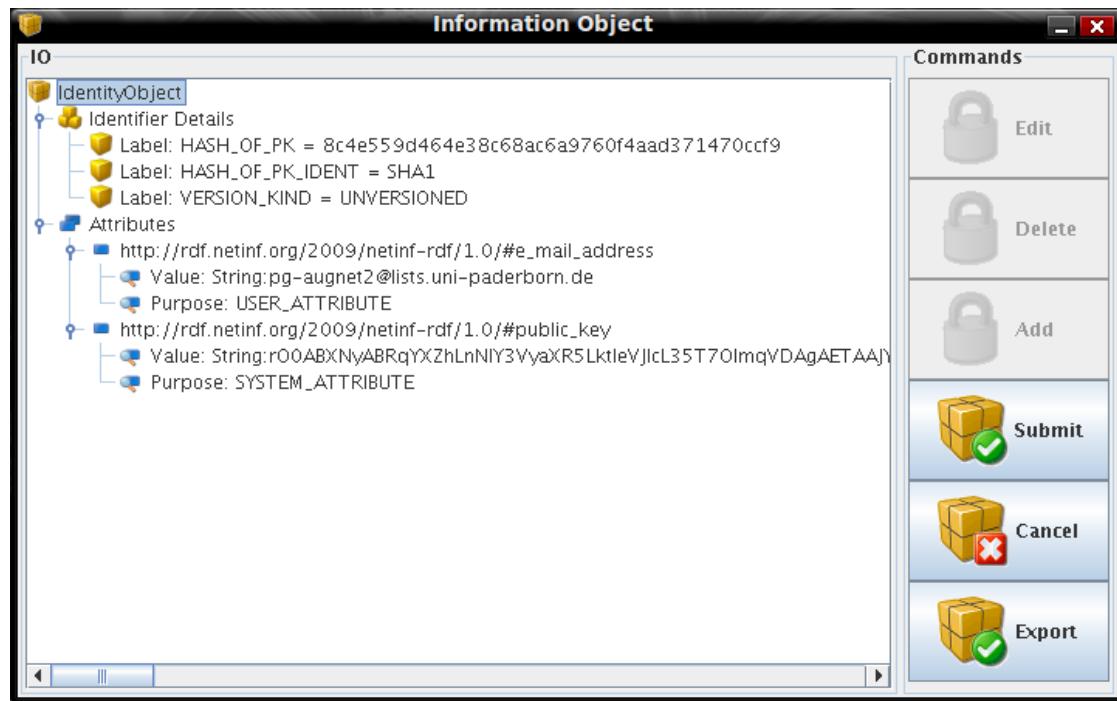


Figure 7.7: IOManagementTool InformationObject window

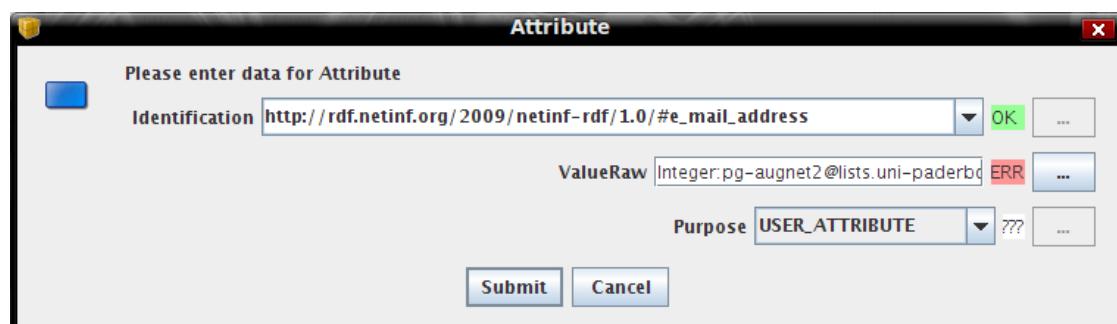


Figure 7.8: IOManagementTool data input window

Chapter 8

Open Issues

The open issues of the project are collected and discussed in this chapter. They may be the starting point for future work.

8.1 Extending Existing Concepts

In this section, extensions for already implemented concepts and open problems of these concepts are discussed.

8.1.1 Data Model

8.1.1.1 Ontologies

Our data model is mainly based on the Resource Description Framework ([RDF](#)) [9]. [RDF](#) is a standard to describe pieces of information, their relations to each other and their context. This way, one can make the semantics of data machine-readable. To describe semantics and express relations one can write an ontology using RDF Schema ([RDFS](#)) [8]. And that is what the prototype misses for now. It is necessary to create this ontology via [RDFS](#) to get full advantage of the data model. Beside adding semantics to the data, an ontology enables more sophisticated search queries. For example, together with the *SPARQL* [11] search it would be possible to infer new relations between pieces of imformation from already defined relations. Furthermore, the integrity of an [IO](#) can automatically be checked, when the [RDFS](#) defines necessary attributes and dependencies between attributes. You can already find a simple version of an ontology in the development repository under `src/InformationObject.rdf`.

There are some [RDF](#) ontology projects that already defined ontologies which could be reused in the prototype. *FOAF*¹, for example, defined an ontology for everything about contacts; *NEPOMUK*² defined an ontology concerning all file formats. Since it makes sense to use those well-defined ontologies, we have to find a way to integrate

¹<http://www.foaf-project.org/>

²<http://www.semanticdesktop.org/ontologies/>

these ontologies in [NetInf](#) Applications. Of course, this should be as easy as possible for the user.

Listing 8.1 shows an excerpt from a possible [IO](#) ontology. Notice that some semantical expressions, such as attribute-requirements and -ranges, require the usage of Web Ontology Language ([OWL](#)) [7], which can be understood as a extension for [RDFS](#). [OWL](#) allows much more sophisticated rules than [RDFS](#) although it is based on [RDF](#). The listing exemplifies on a `http://` attribute how attributes are made mandatory through the [OWL](#) Restriction statement and [RDFS](#) range is used to define which values are permitted. You can simply include [OWL](#) ontologies by adding them into the `rdf:RDF` tag and define a XML namespace. Other ontologies, like FOAF and NEPO-MUK, can be included and used in the same way. The current state of our [IO](#) ontology can be found in Appendix A.

Listing 8.1: Example attribute in an IO ontology

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:netinf="http://rdf.netinf.org/2009/netinf-rdf/1.0/#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">

    [...]

    <rdf:Description rdf:ID="Attribute">
        <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-
            schema#Class"/>
        <rdfs:domain rdf:resource="#InformationObject"/>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#attributePurpose"/>
                <owl:cardinality>0</owl:cardinality>
        </owl:Restriction>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#attributeValue"/>
                <owl:cardinality>0</owl:cardinality>
        </owl:Restriction>
    </rdf:Description>

    <rdf:Property rdf:about="attributePurpose" rdfs:label="
        attributePurpose" rdfs:comment="The purpose of this
        attribute, eg. LOCATOR_ATTRIBUTE, SYSTEM_ATTRIBUTE,
        USER_ATTRIBUTE">
        <rdfs:subPropertyOf rdf:resource="#Attribute"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            FunctionalProperty"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
        <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf
```

```

        -schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="attributeValue" rdfs:label="attributeValue" rdfs:comment="The value of this attribute">
    <rdfs:subPropertyOf rdf:resource="#Attribute"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</rdf:Property>

<rdf:Property rdfs:label="http_url" rdfs:comment="A HTTP URL for this IO">
    <rdfs:subClassOf rdf:resource="#Attribute"/>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#attributeValue"/>
        <owl:allValuesFrom rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    </owl:Restriction>
</rdf:Property>

</rdf:RDF>

```

8.1.1.2 Addressable Attributes

Each **IO** is addressable via its identifier. This should be possible for attributes within **IOs** as well, e.g. via `identifier#attribute-identification`. This would enable pointers to attributes, which is desirable for data consistency reasons. For example, it would be nice to refer to a subidentity in an **IdO**. Note that an **IO** can contain more than one attribute with the same attribute identification. This is the case if a subattribute can be contained in more than one attribute, or if multiple homogeneous values exist for one attribute(-identifier). Thus an **IO** may have more than one attribute `authorized_writers`, if multiple users are allowed to write this **IO**. In that case we like to address a list e.g. of all `authorized_writers` via `identifier#authorized_writers`.

Furthermore, it should also be possible to address subattributes in the same way as attributes, e.g. via `identifier#attribute-identification#subattribute-identification`. You then also have to define how to address all attributes with the same identification within one **IO**. The usage of XPath [2] could be an efficient way to navigate through the attribute-tree within **IOs**.

8.1.2 Security: How to Create an InformationObject That Never Existed

Situation By Now The security approaches implemented in [NetInf](#) up to now guarantee that signed attributes are not added or modified by someone who is not allowed to.

Problem However, the approaches do not guarantee that the [IO](#) that is received – and that has been verified successfully – has ever been stored to [NetInf](#) in that form. Think of the following situation: The owner I of an (unversioned) [IO](#) IO first stored IO with one single attribute A that has been signed. Later on, he removes attribute A and appends one single attribute B that is signed. An attacker may now have seen both versions of the [IO](#). Thus, he knows A and B with its signatures.

Attack The attacker may now put A and B with valid signatures into IO and then send it in response to a get request for IO .

Solution? Using our current approach, the only way to make sure several attributes (e.g., all of the [IO](#)) were stored in that [IO](#) at the same time is to calculate a joined signature (`is_secured_in_overall`). A drawback of this is that you may have to calculate a signature for a large number of attributes instead of only the one you modified.

Furthermore, our current approach includes the option for different parts of the [IO](#) to be signed by different persons, so that different persons guarantee – in a semantical meaning – for the correctness of the information given. In case all signatures have to be recalculated at one attribute's modification, it has to be indicated by other means who is responsible for which content.

Short example: User X knows that a certain kind of mushroom is not poisonous and stores (and signs) this information to an [IO](#). User Y has a new picture of that mushroom and wants to add a locator of it to the [IO](#). Y wants to sign that locator but does not know whether or not that mushroom is poisonous. If now Y signs the information that the mushroom is not poisonous, just because X said so, this may be a problem – as long as signatures somehow imply responsibility for correctness of information.

One might say, these problems may be circumvented by user Y only signing a list of all the attributes that have been present at the moment Y added his locator, instead of actually resigning the whole [IO](#). In this case, the individual signature of user X would remain so that everyone receiving the [IO](#) knows that user X provided the information, not Y .

However, this does not reduce the effort of calculating a new signature for the whole [IO](#). User Y would have to include all relevant pieces of information in this new signature. If Y would sign, e.g., the attribute's identification only, the value of the attribute could still be exchanged. The argumentation is equivalent to Section 3.7.1. In consequence, Y again has to sign all the attributes in the [IO](#).

All in all, this is equivalent to using individual signatures for indicating that someone is responsible for correctness of information, and using `is_secured_in_overall` to indicate that the current combination of attributes has actually been stored by an authorized writer. This concept might be continued consequently by adding a numerical value to the `is_secured_in_overall` attribute. Then all the attributes having the same numerical value in their `is_secured_in_overall` subattribute build a *group* of attributes. Now, instead of only one joint signature for all the attributes marked by a `is_secured_in_overall` subattribute in the whole **IO**, a joint signature is calculated once per group of attributes. Through this addition, several attributes may be grouped such that either all the attributes of the group have to be present, or all have to be removed. User *Y* would be able to calculate a new signature for all the attributes of a certain group of attributes only, instead of all the attributes of the **IO**.

8.1.3 Actuality of InformationObjects

Situation By Now Existing **IOs** can be changed by storing a different **IO** with the same identifier to **NetInf** (i.e., overwriting). Furthermore, **IOs** can be deleted.

Problem However, the **NetInf** prototype does not yet stick to a consistency model³ (not even eventual consistency⁴). Thus, there is no guarantee that someone asking for a certain **IO** gets the version most recently stored to **NetInf**. Even if we would stick to a consistency model, there might be an attacker responding to a get request with an old version of an **IO**. So far, there are no means to detect whether or not a user received the most current version of an **IO**. There is not even a hint.

Attack The problem begins at the heart of **NetInf** security: the **PKI**.

Attributes in **NetInf** are signed (using the private key) by the one who changed the **IO**. For signature verification, the identifier of the respective **IdO** is given, so that the public key can be retrieved. Some author signs an attribute with his or her private key and references his or her **IdO**. Later on, users get the referenced **IdO** and use the included public key to verify the signature.

If now an attacker gets to know the private key related to some **IdO**, the attacker may masquerade as the owner of the **IdO**.

As soon as someone detects that the private key has been disclosed, the public key in the **IdO** should be revoked. Thus, users trying to verify the signature would come by the **IdO** and see that the public key has been revoked. They would not trust the signature anymore.

³A consistency model is something like a promise of the system – in this case **NetInf** – at which point of time a user will see which version of something – in this case an **IO**. Consistency models have been defined in the area of distributed memory and distributed storage.

⁴Eventual consistency is the promise of the system, that eventually – so *at some point of time* – the latest update will be reflected. In case of **NetInf**, eventual consistency would mean that, at some point of time, all nodes will answer get requests for a certain **IO** with the latest pushed version of that **IO** – no matter to which node it was pushed.

However, since there is no possibility of detecting the actuality of an **IO**, the attacker may provide all users with old versions of the **IdO**. There would be no hint about the public key being revoked and the signature would be verified successfully.

Approaches One may relocate the public keys (or the information whether they have been revoked) to a trusted party. Whenever a public key is required, this trusted party would be asked whether or not that public key is still valid. However, this trusted party would need to be contactable all the time. Disconnected operation would be impossible (e.g., at a bus stop, where several people may have **IOs** others are interested in, but where no link to that trusted party is available).

Another approach is that some trusted party – large enough that disclosure of the trusted party’s private key will become public by media (TV, newspapers, ...) – puts signed timestamps into the **IOs**. These timestamps may tell, how long the current version of the **IO** is valid. Regularly, this trusted party would have to confirm that the private key of this **IO**’s owner has not been disclosed, by updating the **IO** with a current, signed timestamp. As soon as the owner of an **IO** notifies the trusted party of a disclosed private key, the trusted party would stop renewing the timestamps.

Drawbacks of the latter approach:

- Although disclosure of private key has been detected, there is a valid **IdO** until timestamp passes by ⇒ delay of revocation
- **IdOs** need to be updated regularly (additional effort)
- Trade-off between revocation delay and update effort is required
- A trusted party is still required

Although this issue is discussed for **IdOs**, the problem to determine the actuality of an **IO** is not limited to **IdOs**, but is relevant for all kinds of **IOs**: If an **IO** about a mushroom once characterized the mushroom not being poisonous, there should be an approach to tell users that there is more topical information on the mushroom in case it has been recognized as actually being poisonous.

The main enabler for the attack discussed in this section is that the content of an **IO** is allowed to change. The same identifier may lead to different content, changing over time. Obviously, introducing a versioning concept that uses a different identifier for each modification of an **IO** is an approach to solve this problem. The author of the **IO** would sign *only one* combination of attributes (i.e., one version) using this version’s identifier. Thus, there is no different valid version an attacker might use for response. However, the requesting user would have to know the identifier of the most current version of the **IO**. This, in turn, requires some *trusted* third party – e.g., a versioning search service – to be reachable all the time and to know the identifier of the most current version of each **IO**.

8.2 Further Concepts

In this section, additional concepts are discussed which are not yet part of our prototype implementation.

8.2.1 Rights Management

Rights management is already implemented in the [NetInf](#) prototype. For information about integrity (write permissions) see Section [3.7.1](#). Reading permissions are described in Section [3.7.2](#). There are, however, further open issues regarding *Groups* and *Ownership Change*.

8.2.1.1 Groups

A GroupObject ([GO](#)) contains a list of [IdO](#) references to persons or other [GOs](#). Thus, a [GO](#) is also an [IdO](#). The group has a common key that is encrypted for each reader with his key (according to the key list for reading permissions). This key list is stored in the [GO](#). Apart from that, a [GO](#) resembles a normal [IO](#) with an owner, etc. Be aware that [GOs](#) have not been implemented yet. [GOs](#) are useful to keep *Authorized Readers Lists* and *Authorized Writers Lists* short.

History of Groups A history of the groups is needed e.g. for the following case: User U is a member of group G . Group G is a member of the *Authorized Writers List* of [IO](#) X . Thus, U is allowed to change X . He does so and creates a new version X_2 of the [IO](#). This [IO](#) X_2 is valid and can be verified. At a later time the owner of G decides to exclude U from the group. If G would only be changed and not include U anymore, X_2 would become invalid although it was legally created. Therefore, a new version G_2 of the group G needs to be created. G includes user U but G_2 does not. With a reference of G in X_2 (since G was the current version of the group when X_2 was created), X_2 can still be verified. Thus, a [GO](#) has to be versioned and references to groups have to be made with the versioned identifier. Each time somebody is removed from a group, a new key has to be created and a new version of the [GO](#) is created.

Remark With this group concept there is the problem that old [GOs](#) can still be used. Has somebody been a member of the group at the time he changed an [IO](#)? Or has he only been a member of the group at some time before he changed the [IO](#)? We need timestamps for every change in [IOs](#) (including changes to [GOs](#)) to answer this question. But neither the members of the group nor the infrastructure should write the timestamps since we should not trust every group member for security reasons. The owner could be in charge of writing the timestamps because he is responsible for his [IO](#) and most likely wants it to be accurate. This would mean that a version of the [IO](#) is only valid if the owner confirms it. But then we would not need time anymore, we simply know whether it is valid or not. This (probably only possible) solution is extremely slow since a change always needs to be confirmed by the owner. Therefore, this solution is not an option.

Thus, we cannot eliminate any possibility of somebody using an old version of a **GO** again. This issue is described in more detail and approaches to solve it are given in Section 8.1.3.

Rights Management of Writing With Groups The *Authorized Writers List* might also contain references to **GOS**. These references have to be the versioned identifier of the **GO**.

If a member of a group changes an attribute of an **IO**, he signs it with the common key of the group. The reference to the according **GO** is stored in the `writer` attribute.

Additionally, there is an attribute called `single_writer` containing the reference to the **IdO** of the writer (single identity, no group) for responsibility issues. For this attribute no identity with children is accepted (everything that does not contain a list with further **IdOs**). This attribute is optional or can be made mandatory by the owner if he sets a corresponding flag called `SINGLE_WRITER_REQUIRED`. This flag expresses whether signatures have to be made by non-groups and whether the attribute `single_writer` has to be set or not. Thus, the owner of the **IO** can decide between anonymity and traceability.

Rights Management of Reading With Groups For each reader group there is an entry *kGroup* in the **IO** with the key *x* encrypted with the key of the group. Key *x* is used to decrypt the **IO**. The label *kGroup* needs to contain the versioned identifier of the **GO** because we need to know the version of the group.

For each version of an **IO** a new key has to be created for encryption because the members of a group who are allowed to read the **IO** might have changed in the meantime. It is not necessary to change the encryption of the previous **IOs**. The user who is no longer a member of the group was once able to read these **IOs**. By that time he could have stored or printed the information from these **IOs**. Therefore, it does not make sense to deny him read access to these **IOs** when he is deleted from the group. It is assumed that if you have had a piece of information once, you have it forever.

8.2.1.2 Owner Change

Each **IO** has an owner. This owner might once desire or be required to give away the ownership. The following paragraphs describe ideas for implementing owner change and its advantages and disadvantages. At the end of this section the concept implemented is explained.

Approach Using Ownership IOs One idea to implement owner change is that for every **IO** that might be handed over, a special **IO** with a key pair has to exist. This special **IO** is called *InformationObjectOwnershipObject* (**IOOO**) in the following. The **IOOO** should refer to the right owner represented by an **IdO** or **GO**. Such an **IOOO** would guarantee that the key pair is bound only to the **IO** itself and not the owner. This is also important since otherwise anonymous **IOs** would not be possible.

When the ownership is handed over, the key of the last owner is revoked which means that it is broken. This ensures that all previous owners no longer have a key which they could use for changing. It is also important for the key to be different from the one of an **IdO**. Otherwise the **IdO** would be unusable afterwards. Therefore, the IOOO is needed.

Furthermore, an owner can use the same key for different **IOs**. Since it is not wanted that all **IO** ownerships with the same key are revoked, but only a specific one, the revocation list should not only contain the key, but also a specific (key, identifier) pair that was revoked.

An advantage of this solution is that the key can be revoked without any influence on the owner and his **IdO**. Additionally, the owner would not have to be stored in the **IO** since it is clear that the last one in the IOOO is the current owner. Whenever an IOOO is revoked, it could also refer to the successor IOOO so that which **IdO/GO** owns it can still be verified relatively fast.

A disadvantage is that this solution depends on a global revocation list. Revocation, by the way, never really worked in praxis, e.g. because browsers do not implement revocation list checking correctly. In any case the key pair should not depend on a trusted third party, this should only be necessary for owner identification.

Another disadvantage is that regarding revocation the trusted third party should have an entry for each key pair that is revoked. A central solution is not possible here, because thus we would build upon a trusted third party that has to keep all revoked keys forever.

Finally – regarding the fact that for every owner a key pair exists and assuming that we have a global revocation list – this solution is nearly impossible to put in practice.

PGP-like Approach A further idea is a PGP-like solution. A new version of an **IdO** is created and this contains an entry that the key pair is invalid and this is signed with the key pair. Whoever receives this version of the **IdO** knows that the key is invalid. Everybody else cannot know it. But since **IOs** do have an origin it is very likely that the newest version can be obtained.

Concept Our solution uses our versioning concept (see Section 8.2.3) and also the PGP-like concept. Revocation lists are not needed.

First, the new owner creates a new version of the **IO** which should be handed over. A new version is necessary because changing the owner changes the identifier as well, since the hashed public key of the owner is part of the identifier. Afterwards, the old owner also creates a new version of the **IO** in which he stores the reference to the new **IO** and signs it. With this **IO** in between, all owner changes can be traced and the current owner can be found.

Thus, owner change is easy doable with versioning. Assume that O is the old owner and N is the new owner. After an owner change the new versions are not allowed to have O as owner. But owner O can still create new **IOs** with a newer version with respect to the version before the owner change. But this is not a problem since it can be proven whether O is an old owner or the current one. To change the owner of an unversioned **IO**, the **IO** has to be made versioned first.

It is necessary that the following is regulated contractually: The old owner has to guarantee that the ownership chain remains valid. Additionally, inheritance has to be assured in case the owner passes away. This is necessary for every owner who wants to hand **IOs** down. Inheritance in **NetInf** means handing over the master key (see Section 8.2.2).

There might be other practicable solutions for owner change. But these all would not allow the public key of the owner to be in the identifier as it is wished for **NetInf**.

8.2.2 Trust: Public Key Infrastructure

To provide owner identification in **NetInf**, a trusted **RS** is needed. Therefore, we assume that at least one **RS** with correct behavior exists. The large number of **RSs** makes this likely. For an **IO** the **RS**, whom the owner trusts and where for that reason the **IO** was primarily registered, can be stored in the attribute `origin`. Thus, it is declared that this **RS** behaves correctly.

For a better understanding of the following paragraphs, the content of an **IdO** is explained further here: An **IdO** contains several public keys. There is one master key that exists only for managing the other keys in the **IdO** and for the hash of the public key in the identifier. The master key itself should not be stored in any other **IO**. The other keys are called subkeys and used for encryption and signing whole **IOs** or attributes of an **IO**.

Owner Identification Owner identification can be done via the master key and the concept of *Web of Trust*. Everybody can add a signature to the master key in the **IdO** including the level of trust by adding a subattribute containing the signature to the attribute containing the master key. It is possible to have trusted authorities (such as your **NetInf** provider) where a user has to identify himself first and which can then verify that a user is trustworthy.

Key Change Subkeys can easily be changed by adding a new public subkey to the **IdO** and not using the old one any further. If the master key is to be changed, for example, if the owner of an **IO** passes away and bequeathes this **IO** to a new owner, a new **IdO** has to be created first. It then has to be written in the old **IdO** that it is invalid. This has to be signed with the old master key. Furthermore, the **IO** has to contain a reference to the new **IdO**. This process is no solution for broken keys, since the old broken key needs to be invalidated additionally. Therefore revocation is needed.

Revocation Revocation needs to be done if a key change is necessary because the owner has lost his key or when others got hold of a private key they were not supposed to know.

Revocation for subkeys is quite easy. The old version of the **IdO** with the broken key is simply invalidated (by adding an attribute to it that indicates this) and a new version with the new key is created.

But the problem here is that revocation for subkeys is only that easy if we can guarantee that after a key has been revoked and pushed to a **RS**, no **RS** may ever return only an older version of the **IdO** that does not contain the information that it is invalid. Thus, users may recognize keys as being revoked only by coincidence.

Furthermore, the `origin` attribute implies a node-centric approach and represents a single point of failure. To make this attribute less important, we have to send get requests for an **IdO** to several **RSs** when checking an **IO** to be able to compare all the returned **IdOs** and to be able to define the desired one. This would require large computation effort.

Losing a master key is really bad and should not happen (comparable to loose your passport *and* your set of keys). But since the master key is only used for managing the other keys, it is not very likely that the master key is lost and gets broken. But if it happens, the old **IdO** has to be invalidated first. This has to be signed with the stolen master key. An invalid **IdO** can never be revalidated and no reference to a new **IdO** is allowed. This can be ensured by implementing that an invalid **IdO** cannot be changed anymore at all. All **IOs** created with this old master key are invalid now. A new **IdO** has to be created, all previous **IOs** have to be collected and the new **IdO** has to be used for creating all these **IOs** new. The new **IdO** could have a reference to the old **IdO** so that it is searchable by a **SS**. This is only a hint for the user.

Open Questions and Problems We assume that at least one **RS** exists, which behaves correctly. We need a trusted **RS** for owner identification. In the **NetInf** prototype trust is not needed so far since owner identification has not yet been implemented. Furthermore, we assume that an `origin` attribute points to such a **RS**. Without this attribute we would have large computational effort as described above. All this implies some open issues, which cannot be solved yet:

1. How do we and how does the user know, which **RSs** behave correctly and which do not?
2. Does the net stand and fall with the **RSs** that are thought to be correct?
3. If we have to rely on certain **RSs**, load balancing and scalability are seriously affected.
4. In a local area as, for example, with three laptop computers next to the user, he cannot do anything without a trustworthy **RS** that is reachable.

8.2.3 Versioning

An **IO** should in general be dynamic and changeable. Since the user usually wants to have the newest version but it should also be possible to get older versions, a revision control is needed in **NetInf**.

NetInf should be able to store and deliver different versions of an **IO** if demanded (unversioned **IOs** might also exist). Furthermore, the user has to be assisted in creating

and changing **IOs**. Each time an **IO** is changed, a new version is created to ensure unambiguousness.

8.2.3.1 Definitions

There are **IOs** that are not under version control. Subsequently, the identifier is called versioned identifier if the **IO** is under version control, otherwise we are just talking about identifiers.

The versioned identifier of an **IO** contains the label `VERSION_NUMBER` and the value of the label `VERSION_TYPE` is `HASH_DSA` for instance. The identifier of an **IO** that is not under version control does not contain the label `VERSION_NUMBER`. The label `VERSION_TYPE` has the value `UNVERSIONED`. For example (largely simplified), if the identifier would be `1234`, two versions of an **IO** could have the versioned identifiers `1234:1` and `1234:2`.

For versioning we have to use the following three attributes of an **IO**: `writer`, `origin` and `parent_version`. The attribute `writer` already exists in our implementation. The attributes `origin` and `parent_version` still need to be added to the **IO** for the implementation of this versioning concept.

The attribute `writer` defines the latest editor of the **IO** and thus the one who also signed the hash of the content. If the `writer` attribute is not there, it was the owner himself who changed the object and created a new version. The writer is represented as the identifier of the **IdO** of the one who created this version.

The attribute `origin` defines where the first version of an **IO** is primarily registered. The value is an identifier of the **IdO** of a **RS**. It is a point of reference where the newest version may be found. The attribute enables deletion: If it is stored at the *original RS*, that an **IO** was deleted, every user can ask this **RS** whether the **IO** still really exist or not. There might be more than one origin because of merging.

The attribute `parent_version` defines the **IOs** of the former version. By knowing the parent versions of all versions of an **IO**, a version graph can be created. An **IO** without the `parent_version` attribute is the root of such a version graph. **IOs** with more than one `parent_version` attribute are results of merges. The `parent_version` attribute can contain any valid identifier of **IOs**.

Hence, an **IO** contains information about the history of the **IO** with a `parent_version` attribute that can exist multiple times and contains the versioned identifier for **IOs** that this one is based on. For example (simplified by using incremental version numbers), if there is an **IO** `x:1` that is a new version of **IO** `x:0`, then `x:1` contains a `parent_version` attribute with value `x:1`. If there is another **IO** `x:2` that is a merge of `x:0` and `x:1`, then `x:2` contains two `parent_version` attributes with value `x:1` and `x:0`. This also allows non-owners to create derivates. For example, the **IO** `x:1` is owned by *X* and *Y* wants to create a derivate and *Y* is not allowed to write `x:1`. Then the derivate will be called `y:0` and contains `parent_version x:1`.

We do not need a pointer to the following version since they can be build with the help of `parent_version`. Current version control systems such as *Mercurial*⁵ only

⁵<http://mercurial.selenic.com/>

store the previous version as well.

8.2.3.2 Use Cases

The following use cases have to be considered regarding a versioning concept for IOs:

Create New IOs: A new namespace is created together with a first version.

Create New Versions: Only an owner or someone who has write access is allowed to create a new version. The signature that is stored in the identifier is signed by the writer. Whether someone is authorized or not is looked up in the attribute `authorized_writers` in the IO of the `parent_version`. The right to write of user *A* can be withdrawn in the following way: The owner creates a new version that does not contain user *A* in `authorized_writers` anymore.

Create Derivates: If others than the owner or authorized writers want to use a specific IO and extend it, they can create derivates. Then, a new namespace *Y* is created together with a new first version. Additionally, the attribute `parent_version` contains the identifier of the used IO from the other namespace *X*.

Validate Derivates: The owner of the original IO within the namespace *X* can make the new IO of namespace *Y* valid in his own namespace *X*. For this he simply creates a new version inside his own namespace, that points to the version in namespace *Y*. This is a special case of a merge.

Merge Two Versions: This use case should be done through an existing version control system that has to be embedded.

8.2.3.3 Concept

In the following, different aspects of a concept for the versioning of IOs are described.

Embedding of Version Control Systems We care about data storage whereas everything else is done by other version control systems as an additional service. This means that we only name the versions and connect them via the `parent_version` attribute. Complex version control such as tracking and merging is done by specific systems (e.g. *Mercurial*).

The interface to the version control systems should allow retrieval of versions, retrieval of the version graph and creation of versions. An implementation of such an interface prohibits the version numbers from being monotonically increasing because otherwise there needs to be a central instance that assigns new version numbers. This is one of the reasons why we use the hashed content as the version number.

An important advantage of the embedding of an existing version control system as an additional service is that a next generation version control system can be embedded easily. Additionally, such an implementation cannot lead to implementation specific problems with different control systems because the only job they are doing is creating new versions based on other versions.

Representation of the Version Number We thought of representing version numbers either as a hash, a timestamp or consecutively by monotonically increasing numbers.

Our concept suggests the usage of the hash as version number where the version number is represented through the hash of the content. We do not need to take the signature of the content as the version number since this only implies additional computation effort and does not bring any advantage to the security because the rights to write are already ensured with other mechanisms (see Section 8.2.1).

We also offer to use other schemas for version numbers. Version control systems can append their version scheme and the semantic is controlled by them. An option would be to put sequential numbers after the hash. The sequential number could also be assigned by the user. In principle, the user can add anything after the hash. Thus, the semantic can be controlled by the user. This could also be used for branches: equal sequential number are used, but the hash is different. The type of the version number is to be defined in the identifier as it is described in the next paragraph.

Another option would be to use timestamps. This has an advantage over hashes, namely that newer versions are directly identifiable. But uniqueness is a problem and timestamps do not identify the content in any way. Another problem with timestamps is that there should no global time be used for this in [NetInf](#). The user has responsibility which could be problematic (wrong clocks etc.).

Version Number as a Part of the Name As already mentioned at the beginning of the chapter, we store the version number in the identifier of an [IO](#). We decided the version not to be an attribute of the [IO](#) because it is a conceptional part of the name. A name identifies a specific information and a new version of an information is actually a different information. Thus, different versions should have different names.

Furthermore, asking for specific versions is complex. A resolver needs to be asked for a specific [IO](#) name and then somehow it has to be instructed to only return a specific version. It then has to look into the [IO](#)'s attributes to get the correct version. In contrast, if the version number is part of the identifier, it is easy to refer to a particular version with a known version number. Each [IO](#) that differs has an own identifier.

The versioned identifier thus is of the form <NAME><VERSION>. VERSION includes the label VERSION_NUMBER and VERSION_TYPE, NAME includes all other labels of the identifier. This allows easy requests for a specific version of some [IO](#). NAME identifies a collection of documents with different versions and VERSION is e.g. a hash over the complete content of the [IO](#) in that single version. Thus, each change to an [IO](#) creates a new version.

Generic [IO](#) requests will omit the VERSION part. The resolver will then return the newest version of the [IO](#) that it knows by default. It is also possible for the resolver to return all versions and a Version [SS](#) or the user himself selects the newest version of the returned [IOs](#). New versions of an [IO](#) can only be created by the owner(s) or others with the right to write the [IO](#).

Name Resolution There are two cases to request an [IO](#). The first is that one requests an [IO](#) with the complete identifier (the name and the version). This simply works as it does now with the [RS](#).

The second case is to request all [IOs](#) with the NAME-part of the identifier alone.

Thus, we would like to get all available versions. Or we could possibly only ask for specific ones such as the ones with a specific predecessor. The resolution in this case can be done with the help of a Version **SS**. The Version **SS** stores the version graph. We assume that the Version **SS** can build the graph efficiently. The Version **SS** needs to know the identifier, the `parent_version` and the `origin` attributes of each **IO**.

A resolver needs the following special query capabilities:

1. To get an **IO** / the **IOs** that are created by some person and have a specific previous version: REQUEST `io WHERE ATTRIBUTE netinf:owner <someidentity> AND ATTRIBUTE netinf:previous_version <somename>:<someolderversion>` (There could also be a transitive previous version query)
2. REQUEST `io WHERE ATTRIBUTE netinf:previous_version <somename>:<someversion>` (This would also give the `y:0` in the above example)

8.2.3.4 Appraisal of the Concept

With our concept it is possible to create a version control system with **NetInf** that uses the versioning and version graph features and, additionally, provides easy merging, commit messages (special attributes), etc.

It is not possible to know all versions of some **IO**. It is only possible to get all versions that can be reached due to the open world assumption. This is no difference to what we have now. It is impossible to know if an **IO** exists or not. This also means that some previous versions of an **IO** might be unreachable or the version control system would need to store the complete history to ensure that this does not happen.

Robustness We still have to assume that clients have incorrect behavior that is described in the following three cases:

Case 1: A client creates a circular previous version graph. This cannot happen as the version is defined as the hash over the content. A collision is almost impossible (even less possible than for the `NAME` part) and a collision is detected by the client by fixing this on creation of a new version. Therefore the user simply updates the creation date of the version by X and the hash changes. If there still is a collision or intentional circle the **IO** that does back-referencing is broken and cannot be used in a useful way. All other **IOs** in the circle are still correct, valid and usable.

Case 2: A client creates an **IO** with a `parent_version` attribute that points to random, unrelated other **IOs**. This only means that the history information of the **IO** is broken and has no wider implications.

Case 3: An invalid version is used for an **IO**. This is no problem because if the **IO** is received a conflict between the hash of the identifier and the content exists and the **IO** can be detected as invalid.

It also doesn't pose a problem if a user receives an incomplete graph. If he would then work on an old and not a new version, this could easily be solved by another user by creating a new version that merges the version he created with the newest version.

Length of the Name The name actually might get rather long but this is not critical in regard to good performance. In the NAME part the maximum length of the specific labels could be defined by the infrastructure. In the VERSION part the length of the signature depends on the hash function. Thus, the length of the signature is limited by the function.

8.2.4 Collaborative Caching

The caching components of NetInf (see Sections 3.4.4 and 3.6.4) already allow data to be migrated to nodes close to regions with high congestion automatically. Neighbors of requesting nodes in this region are likely to have cached the data because they needed it themselves. However, currently nodes only maintain a cache on their own. They might replace data they do not need but that is still important for their neighborhood. If many neighboring nodes are interested in the same data they might all cache this data while other data that is still of interest for some of the nodes is not cached due to lack of space. The network might benefit from collaboration in the decision of what data is cached.

An approach could be making nodes use a distributed RS as a cache. The architecture already allows to use any RS implementation for caching (see Section 3.4.4). The distributed cache could replace the local cache or be an addition to it. The RS might be similar to the Pastry RS (see Section 3.4.2) but should only connect a small neighborhood of nodes. Since this distributed cache has information about all nodes that use it, it could define how many copies of an IO have to exist and decide which IOs are replaced, based on the overall popularity in the neighborhood.

Before caching a BO the node could use the *Locators* listed in the respective DO to check whether the BO is already cached in its neighborhood. This information might decrease the importance of caching the BO. A BO not being cached in the neighborhood could also become less likely to be deleted to guarantee diverse cache content. A distributed storage engine like Pastry might be applied to store BOs as well.

Acronyms

BO BitlevelObject

DO DataObject

ES Event Service

ESF Event Service Framework

GO GroupObject

GP Generic Path

IdO IdentityObject

IO InformationObject

MDHT Multilevel Distributed Hash Table

NetInf Network of Information

OWL Web Ontology Language

PKI Public Key Infrastructure

RC Resolution Controller

RDF Resource Description Framework

RDFS RDF Schema

RS Resolution Service

SC Search Controller

SS Search Service

TC Transfer Controller

TS Transfer Service

TD TransferDispatcher

DHT Distributed Hash Table

Bibliography

- [1] 4WARD Consort. 4WARD – Architecture and design for the future Internet, 2008.
- [2] James Clark and Stefe DeRose. XML path language (XPath). W3C recommendation, W3C, November 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116/>.
- [3] C. Dannerwitz, J. Golic, B. Ohlman, and B. Ahlgren. Secure naming for a network of information. In *13th IEEE Global Internet Symposium 2010*, 2010.
- [4] M. DAmbrosio, C. Dannerwitz, H. Karl, and V. Vercellone. Mdht: a hierarchical name resolution service for information-centric networks. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pages 7–12. ACM, 2011.
- [5] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- [6] R.T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, Citeseer, 2000.
- [7] W3C OWL Working Group. OWL 2 web ontology language document overview. Technical report, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>.
- [8] Ramanathan V. Guha and Dan Brickley. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [9] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.
- [10] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCS Workshops*, pages 611–618, 2002.
- [11] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.

Appendix

Appendix A

Data Model – InformationObject Ontology

This appendix provides our current ontology for IOs. The ontology contains all attributes defined in netinf.common.datamodel.attribute.

DefinedAttributeIdentification:

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#
"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdfs:ID="InformationObject">
    <rdfs:comment>NetInf InformationObject</rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://www.w3.org
        /1999/02/22-rdf-syntax-ns#Resource" />
</rdfs:Class>

<rdfs:Class rdfs:ID="DataObject">
    <rdfs:comment>NetInf DataObject</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#InformationObject" />
</rdfs:Class>

<rdfs:Class rdfs:ID="IdentityObject">
    <rdfs:comment>NetInf IdentityObject</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#InformationObject" />
</rdfs:Class>

<rdf:Property rdf:ID="ioType">
    <rdfs:comment>The InformationObject's type, e.g. DataObject
    </rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
```

```
</rdf:Property>

<rdf:Property rdf:ID="attributePurpose">
    <rdfs:comment>The attribute's purpose: SYSTEM, LOCATOR or
        USER</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="attributeValue">
    <rdfs:comment>The attribute's value</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="owner">
    <rdfs:comment>Identifier of the IO's owner</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="parent_version">
    <rdfs:comment>In a versioned IO, the Identifier of the
        parent version of this IO</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="origin">
    <rdfs:comment>Indicates a (sort of) primary Resolution
        Service this IO is stored at</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="branch_name">
    <rdfs:comment>In a versioned IO, the name of the versions
        branch</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>
```

```

<rdf:Property rdf:ID="time_to_live">
    <rdfs:comment>Estimated point in time when a newer version
        of the IO will be available</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="authorized_writers">
    <rdfs:comment>Identifier of IdentityObject of entities that
        are allowed to modify this IO</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="authorized_readers">
    <rdfs:comment>Identifier of IdentityObject of entities that
        are allowed to read encrypted content of this IO</
        rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="writer">
    <rdfs:comment>Identifier of IdentityObject of the entity
        that signed the IO</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="writer_of_group">
    <rdfs:comment>Not used yet. Identifier of IdentityObject of
        the entity that is authorize to change IOs in the name
        of the group</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="signature">
    <rdfs:comment>Contains the signature of the IOs content to
        check its integrity</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />

```

```
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="signature_identification">
    <rdfs:comment>Indicates the Signature Algorithm used to
        sign the IOs content</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="signature_verification_failed">
    <rdfs:comment>Indicates that the Signature Verification
        failed</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="hash_identification">
    <rdfs:comment>The hash algorithm's name for data hashing</
        rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="hash_of_data">
    <rdfs:comment>The hash of the BO, based with
        hash_identification</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="person_name">
    <rdfs:comment>If the IO is an IdO representing a person,
        this attribute contains the person's name</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="person_age">
    <rdfs:comment>If the IO is an IdO representing a person,
        this attribute contains the person's age</rdfs:comment>
```

```

<rdfs:domain rdf:resource="#InformationObject" />
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="e_mail_address">
  <rdfs:comment>E-Mail address, e.g. for InBird or InFox</
    rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="members_of_group">
  <rdfs:comment>Not used yet. Identifier of IdentityObject of
    the entity that are members of the group</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encrypted_group_keys">
  <rdfs:comment>Not used yet. Keys a group uses to read/sign
    IOs</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="cache">
  <rdfs:comment>true if the locator is cached</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="identity_revoked">
  <rdfs:comment>Indicates that a Private-/Public-KeyPair in
    an IdentityObject has been revoked and that it should
    not be used to validate signatures any more</
    rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="third_party_signature">
```

```

<rdfs:comment>Not used yet. Signature done by an Identity
other than writer or owner. Will be used for Web of
Trust.</rdfs:comment>
<rdfs:domain rdf:resource="#InformationObject" />
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="single_writer_required">
    <rdfs:comment>Not used yet. States whether or not a writer
has to add an Identifier of his own IdentityObject when
signing in the name of a group.</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="identifier">
    <rdfs:comment>The IOs Identifier</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encrypted_information_object">
    <rdfs:comment>Contains a whole IO in a symmetrically
encrypted version (used when a whole IO has restricted
read access)</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encrypted_content">
    <rdfs:comment>Contains an Attribute in a symmetrically
encrypted version</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encrypted_reader_key_list">
    <rdfs:comment>Parent element of all encrypted reader keys</
    rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />

```

```

</rdf:Property>

<rdf:Property rdf:ID="encrypted_reader_key_entry">
  <rdfs:comment>A reader key entry for one Identity
    authorized to read. It has the path to the public key
    used for encryption as value. It contains the encrypted
    reader key for this Identity.</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encrypted_reader_key">
  <rdfs:comment>A reader key for one Identity authorized to
    read. It has the the reader key for this Identity in an
    encrypted version as value.</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="encryption_algorithm">
  <rdfs:comment>Indicates the algorithm used to encrypt the
    IOs content</rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="public_key">
  <rdfs:comment>The PublicKey of the Public-/Private-KeyPair
    used to proof ownership of an Identity (IdentityObject)
    </rdfs:comment>
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="preferred_reader_key">
  <rdfs:domain rdf:resource="#InformationObject" />
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
    syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="secured_in_overall">
  <rdfs:comment>Indicates that an Attribute takes part in the
    IO wide Signature</rdfs:comment>

```

```
<rdfs:domain rdf:resource="#InformationObject" />
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="description">
    <rdfs:comment>Can be used to describe the object that is
        represented by the IO</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="event">
    <rdfs:comment>Used by the Event Service to store events
        when the subscriber is offline</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="default_priority">
    <rdfs:comment>Default priority value for a
        ResolutionService</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="delete">
    <rdfs:comment>This attribute determines that an IO is
        deleted. Then only the IO-resource including this
        attribute is allowed to be present</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="name">
    <rdfs:comment>The name of the object that is represented by
        the IO (regarding Scenario 2: e.g. the name of a
        represented shop)</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#Literal" />
</rdf:Property>
```

```

<rdf:Property rdf:ID="identity_verification_failed">
    <rdfs:comment>Indicates that the verification of a certain
        Identity being allowed to write to this IO failed</
    rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="http_url">
    <rdfs:comment>A HTTP url for referencing information
        outside NetInf. Used by InFox.</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="represents">
    <rdfs:comment>Used in Scenario 2 to distinguish Shop IOs
        and ShoppingList IOs </rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="product">
    <rdfs:comment>A products name (especially used in Scenario
        2)</rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

<rdf:Property rdf:ID="amount">
    <rdfs:comment>Indicates how many units of a certain product
        are present in a shop (especially used in Scenario 2)</
    rdfs:comment>
    <rdfs:domain rdf:resource="#InformationObject" />
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#Literal" />
</rdf:Property>

</rdf:RDF>

```


Imprint

The logo contains Crystal Icons (<http://everaldo.com/crystal/>) by Everaldo Coelho, licensed under LGPL.