University of Paderborn

Computer Networks Group

FIT: Future Internet Toolbox — Extended Report

Thorsten Biermann, Christian Dannewitz, and Holger Karl

{thorsten.biermann|christian.dannewitz|holger.karl}@upb.de

February 2010

Technical Report TR-RI-10-311

Technical report

Abstract

Prototyping future Internet technologies is an important but complicated task, mainly caused by incompatibilities to existing systems and high implementation complexity. To reduce these problems, we have developed the *Future Internet Toolbox (FIT)*, consisting of four frameworks that cover data transport, information-centric networking, naming, and name resolution. These frameworks can be used separately or can be combined to a large testbed, covering many aspects at once. Experience has shown that FIT not only simplifies our own prototyping activities but is also useful for other projects due to its generality.

1 Introduction

Currently, a lot of research is done in the area of future Internet technologies. Hot topics include information-centric networking, data transport techniques and protocols, routing schemes, resource allocation, mobility, and cooperation/coding techniques. After developing new concepts and protocols in these areas, they have to be evaluated. Often, it is desirable to do this via prototyping and testing under real-world assumptions in real networks. This procedure, however, is very difficult today and is often avoided because (1) the new concepts are incompatible with existing systems or (2) it is too costly to implement a whole prototype from scratch to evaluate a small component of an overall architecture.

To overcome these difficulties, we developed *FIT* – the *Future Internet Toolbox*. FIT simplifies developing future Internet prototypes and testbeds by providing a set of frameworks, covering the following aspects of networking:

- Data transport, including related aspects like routing, resource management, mobility, and cooperation techniques
- Naming and name resolution
- Information-centric networking, including search, publish/subscribe, caching/ storage, and information modeling

These frameworks solve both problems mentioned above as they (1) provide generic testbeds that integrate into today's network/system architectures and (2) provide a lot of ready-to-use building blocks that support and ease development. The downside of such high reusability is that frameworks always introduce a trade-off between reusability and flexibility. Having this in mind, the main goal was to mitigate this trade-off by designing the frameworks as generic as possible to allow implementing as many different concepts and protocols as possible.

We developed these frameworks in the 4WARD research project [1] to build a complex prototype, called NetInf [2]. The prototype focuses on information-centric networking and specialized data transport to fully unfold the benefits of the information-centric network paradigm. This prototype demonstrates that our frameworks can easily be integrated with each other to exploit synergies.

The following sections introduce the frameworks that constitute FIT. Section 2 covers information-centric networking, Section 3 our naming framework. Data transport is addressed in Section 4 and Section 5 discusses generic name resolution.

2 Information-centric networking framework

This section will describe the Information-Centric Network (ICN) framework. We will first describe the framework itself and will afterwards illustrate how the framework can be used in specific information-centric architectures using the example of NetInf and Content-Centric Networking (CCN) [3].

2.1 Overview

Several information/content/data-centric network architectures have been proposed lately [2, 3, 4]. To support the prototyping of ICN concepts, a framework providing the following aspects is desirable:

- A generic, adaptable *node structure* implementation
- A flexible, reusable implementation of various *architecture components*
- Support for defining *new services and protocols*, incl. especially communication

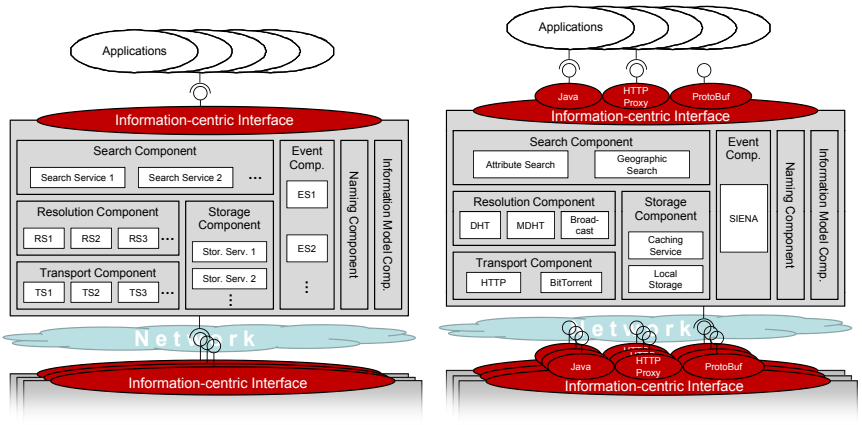
Multiple testbeds like OneLab [5] exist that offer an important environment to test ICN architectures. Those testbeds, however, do not offer support for developing new architectures in the first place. There are generic architectures like ANA [6] that offer a framework for developing and adapting future Internet architectures. But the scope of ANA is too broad to provide practical and sufficient support for prototyping new ICN architectures. There are also specialized middleware projects in the area of content-centric networking like Juno [7]. Juno provides an adaptable middleware for content networking that automatically (re)configures itself based on certain heterogeneity factors. It is an example of a specific, adaptable architecture of an information-centric middleware that could benefit from the existence of a flexible framework for prototyping ICN architectures in general, but does not provide such a framework itself.

We are not aware of a framework that focuses on prototyping ICN architectures. Therefore, we developed the ICN framework for rapidly and easily implementing and testing ICN designs as well as specific services and protocols.

The ICN framework solves the apparent conflict between flexibility and reusability by recursively addressing this conflict on four different levels: the *network architecture level* including communication between network nodes, the architecture of each *network node* that is composed of components, the architecture of each *component* containing multiple component services that implement architecture-specific services and protocols, and the design of those *component services*. On each level, we provide reusable structure and building blocks to accelerate the prototyping process while at the same time providing flexibility and extensibility based on a consistent plugin concept.

Figure 1a gives an example of a node structure for a generic ICN node. The node is composed of several components. Each component can contain a single (see *Naming*, *Information Model* component) or multiple different implementations of its component functionality (e.g., *Resolution* component).

Applications access an information-centric node via its adaptable interface. The same interface can be used to communicate with other information-centric nodes over the network, as illustrated at the bottom of the figure. Inter-node communication and applications will typically access the interface in different ways. To provide a broad and flexible mechanism to access the interface, we have implemented a layer of indirection on top of the interface (Figure 1b) that provides access in different ways (e.g., via an



(a) Generic node structure

(b) NetInf-specific node configuration

Figure 1: Example testbed node configurations, consisting of adaptable components that each contain several service incarnations like *Resolution Services* (*RS*), *Transport Services* (*TS*), and *Event Services* (*ES*).

HTTP proxy interface to connect legacy applications) and can easily be extended with new mechanisms.

In the ICN framework, an ICN can consist of any number of diverse, custom-built information-centric nodes as well as other, non-information-centric nodes. Thereby, we provide flexibility in the design of an ICN architecture. In the following, we will first discuss how to build information-centric nodes with self-defined functionality. Later on, we will discuss how to interconnect nodes to an ICN.

Based on our own conceptual work on the NetInf prototype [8] and evaluation of related concepts like Content-Centric Networking [3], Content-Based Networking [9], and DONA [4], we identified the following main components of an ICN architecture: *Search*, *Naming*, *Name Resolution*, *Data Transport*, *Storage*, an *Event Service*, and an *Information Model*. The ICN framework provides ready-to-use implementations for those components.

Each component can be adapted with architecture-specific services and protocols based on a flexible plugin concept. Thereby, services and protocols are encapsulated in *component services*. To enhance flexibility, a component can contain multiple component services while each service fulfills the same interface but may implement and fulfill this service in a different way. A *component controller* is responsible for choosing between component services, and managing the order of execution. For example, the *Resolution Controller* manages several *Resolution Services* that implement specific name resolution mechanisms, e.g., via a DHT system or via DNS, that can be chosen depending on the type of namespace (flat/hierarchical) to resolve. Adding a new service to a component is done by simply plugging the new component service into the specific component and reconfiguring the controller to include the new service.

To support implementing architecture-specific services and protocols, the other three frameworks included in the FIT can be used to develop custom-built component services for the Naming component (Section 3), Transport component (Section 4), and Name Resolution component (Section 5), illustrating the simplicity of integrating the different FIT frameworks.

Besides the architecture of a network node, the communication between nodes is the second main aspect of an ICN architecture. The ICN framework offers two distinct components for implementing and testing various communication protocols. First, the *Transport* component is used to implement and test communication protocols in general. Several implementations for legacy protocols are already provided by the ICN framework and new protocols can be implemented using the data transport framework (Section 4). Second, as the publish/subscribe paradigm often plays an important role in ICN [9, 2], we provide a dedicated *Event* component to simplify the integration of different publish/subscribe mechanisms into the architecture.

In summary, the ICN framework provides a testbed for building custom nodes and for interconnecting those nodes to implement and test different ICN architectures. It has a flexible and adaptable structure, and offers ready-to-use implementations for the main components of many ICN architectures. It also includes multiple component service implementations for those components to accelerate the prototyping process.

2.2 Implementation

The ICN framework is consistently based on the interface design pattern for flexibility. It is implemented in Java to allow for platform independence and a flexible choice of devices. Special attention has been paid to the selection of libraries to make the framework usable also on mobile devices. The code currently works on FreeBSD, Linux, Windows, and Android.

The plugin concept is based on Google's lightweight dependency injection framework *Guice* [10]. Guice is used for two specific purposes. First, we use Guice for constructing an information-centric node from multiple components, making the overall node architecture easily extensible and specific component implementations exchangeable. Second, Guice is used to plug component services into components to extend them with new service implementations.

2.3 NetInf use case

Figure 1b shows the configuration of a NetInf node created with the ICN framework. Each component contains several specific service implementations that represent the main services and protocols of the NetInf architecture. For example, the NetInf architecture contains multiple name resolution services in parallel that are each represented as a *Resolution Service* (*RS*) in the *Resolution* component.

The NetInf prototype also illustrates the generality of the ICN framework by integrating the NetInf architecture with another content-based network architecture,

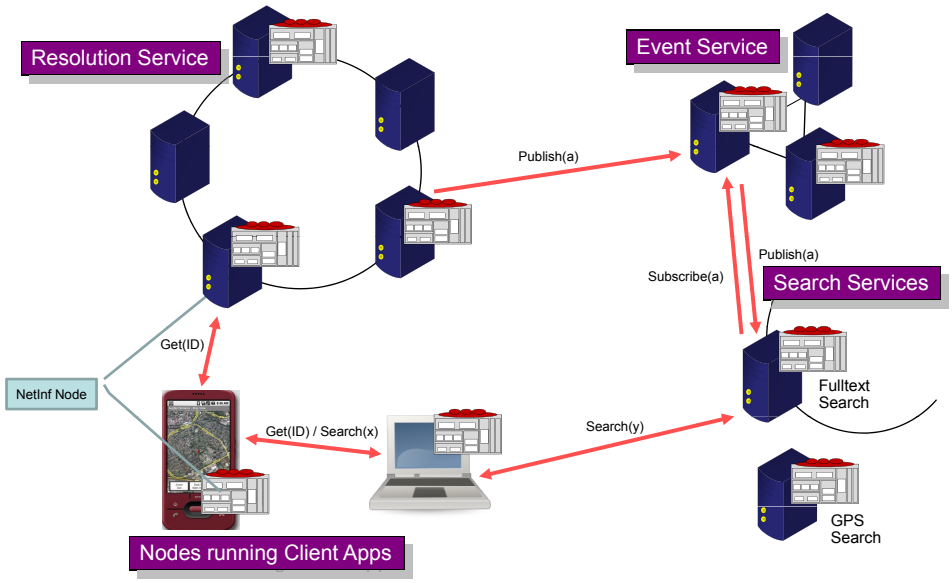


Figure 2: Network of information-centric nodes

SIENA [9]. SIENA is used to realize the NetInf-specific publish/subscribe communication, integrated with the NetInf information model.

Our NetInf prototype extends the information-centric node interface with three different mechanisms to access the node interface (Figure 1b): Java applications can simply use the provided Java interface. For applications that talk HTTP (e.g., a Web browser plugin), we provide an HTTP proxy interface. For inter-node communication, we provide access to the node interface via Google Protocol Buffers (Protobuf) [11]. Google Protobuf enables the simple and fast definition of custom protocol messages and provides efficient data transfer. Many different languages like Java, Python, and C++ are supported, thereby also enabling communication between network nodes written in different languages.

Figure 2 shows an example of several NetInf nodes that form an ICN. Some are running client applications while others provide services like global name resolution and search services. Communication between those nodes is based on the information-centric node interface (implemented with NetInf-specific primitives like *Get(ID)*, *Publish(a)*) and Google Protobuf.

2.4 CCN use case

A prominent proposal for an ICN architecture is the Content-Centric Networking (CCN) [3] project. In the CCN architecture, there are at least three different types of node: clients, servers, and routers. All three node types can be prototyped using our generic node architecture (Figure 3). The key component of the CCN router node is the CCN forwarding engine. It consists of two tables (*Pending Interest Table* (*PIT*) and *Forwarding Information Base* (*FIB*)) that are used for the CCN-specific forwarding

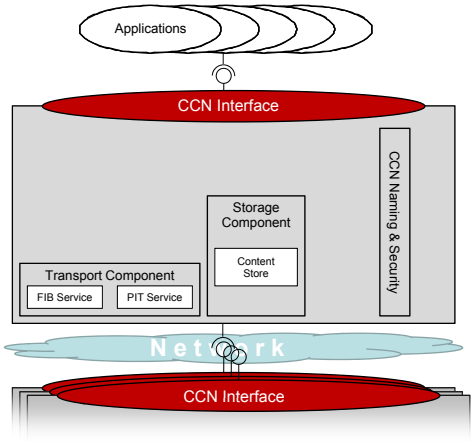


Figure 3: CCN architecture implemented using the ICN framework node architecture

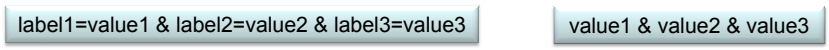
algorithms. Both algorithms and corresponding data structures can be implemented as a *Transport* component service. The third component of the CCN forwarding engine is the *Content Store* which caches data packets in the router. The *Content Store* can be implemented as a *Storage* component service that closely interacts with the afore mentioned *Transport* component service.

Another important part of the CCN architecture is the CCN naming scheme and CCN's data-centric security mechanisms. Naming and security will be an aspect of router nodes as well as client and server nodes. The CCN naming scheme can be modeled with the naming framework (Section 3) and can be integrated in our node architecture via the naming component. The same component will include the CCN security functionality, especially the mechanisms for "validating content" and "establishing trust" [3].

3 Naming framework

A naming framework should provide mechanisms that support a wide variety of different naming schemes while at the same time minimizing the implementation overhead. As a result of those considerations, our naming framework is based on a simple, yet flexible and powerful mechanism: names are composed of a sequence of labels, each of which is a 'labelName=labelValue' pair. There are two ways to handle the ordering of labels: *labelNames* are either themselves part of the name (Figure 4a), thereby explicitly assigning *labelValues* to *labelNames*, or *labelNames* are not part of the name (Figure 4b), which then requires to define the ordering of labels in advance.

Including the *labelNames* in each name allows for flexibility. This makes it possible to define naming schemes with a flexible set of labels as well as a flexible ordering of labels. On the other hand, namespaces with compact names can be achieved by excluding *labelNames* from the names and predefining an explicit label ordering. Via those two mechanisms, our naming framework supports common naming schemes like



(a) Name including label names (b) Name excluding label names

Figure 4: Names consist of a set of labels

IP addresses and URIs as well as complex naming schemes like the NetInf naming scheme [2].

Some more complex naming schemes, especially in the area of information-centric networking [2, 3], involve features like information-centric security that go far beyond common naming schemes. To support such features, we optionally combine names with a flexible set of metadata that can, e.g., contain security-related data like a hash value of the content. In addition, our framework integrates implementations for security-related algorithms like symmetric and asymmetric encryption, (self-)certification, and public/private-key-based authentication to simplify the implementation of complex naming schemes which include such security features.

4 Data transport framework

4.1 Overview

The main observation that triggered our work to develop a data transport framework was the difficulty to introduce new functionality/protocols into today's network stacks. One reason for this is that there is no coherent way to identify entities and to manipulate them as today's network architecture is based on a mainly statically layered stack and functionality is located in end systems.

To design a more flexible, powerful, and reusable data transport architecture, we need an approach to model, design, identify, and use data flows. This approach, however, needs to be generic enough to stretch over a wide range of technology levels and should encompass a wide range of data processing and forwarding functions in end systems as well as in intermediate nodes.

With such a set of requirements, it is impossible to come up with *the, single* solution for a *one-size-fits-all* flow type. Therefore, we decided to choose a design process that combines (1) a uniform appearance and interface for all different flow types and (2) flexibility in supporting a wide range of flow types, in as many different environments as possible. A network architecture that fulfills these requirements is the *Generic Path (GP) architecture* [12]. We use its concepts as basic building blocks for our data transport framework.

We chose an object-oriented approach to design network components while keeping them coherent in their interfaces and basic structures. This allows to incorporate new networking techniques more flexibly than in today's network architectures as networks can be arbitrarily composed of the components. Furthermore, networks can easily be

adapted according to any cross-layer information during runtime, thanks to the unified interfaces. Examples for data transport aspects that have been modified/integrated into our framework are routing, mobility [13], cooperation and coding techniques [14], and resource allocation.

4.2 Framework components

This section introduces the components that constitute the data transport framework: Generic Paths (GPs), Entities, Compartments (CTs), Endpoints (EPs), and Hooks. An example of their interactions is given in Figure 5. The scenario consists of six Entities (E1-E4, two Cores), four CTs (C1, C2, N1, N2), four EPs (EP1-EP4) forming two GPs, and two Hooks. The GP between EP1 and EP2 in C1 (e.g., IP) is realized by the GP between EP3 and EP4 in C2 (e.g., Ethernet).

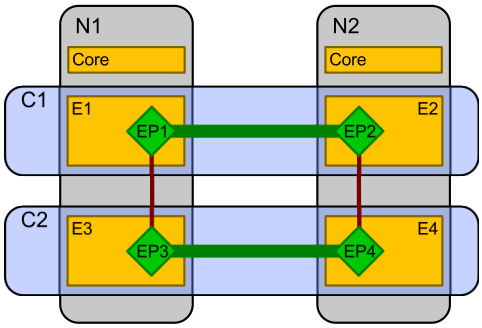


Figure 5: Overview and interaction of GP architecture components. Entities are drawn as rectangles, CTs as rectangles with rounded corners, EPs as squares, GPs as horizontal lines, and Hooks as vertical lines.

4.2.1 Entity

An Entity is the generalization of an application that takes part in any kind of communication. Depending on the implementation, this can be a process, a set of processes, a thread. Communication between Entities is realized via GPs.

Furthermore, an Entity keeps state information that is shared among multiple GPs and runs processes or threads that manage this state. Examples for such state information are routing tables, resolution tables, and access control tables.

4.2.2 Generic Path

A Generic Path (GP) is an abstraction of data transfer between communicating Entities located in the same or in remote nodes. The actual data transfer, including forwarding and manipulation of data, is executed by EPs.

4.2.3 Endpoint

An Endpoint (EP) keeps the local state information of a specific GP instance, i.e, it is a thread or process executing a data transfer protocol machine and doing any kind of traffic transformation. EPs are created by an Entity and may access shared information of that Entity.

Usually, GPs require other GPs to provide their service. E.g., a TCP/IP GP requires another GP that provides *unreliable* unicast, like an Ethernet GP, to provide a *reliable* unicast service at the end. Therefore, EPs are bound via Hooks to other EPs within the same node. Besides exchanging data, Hooks also hide names from other CTs to permit changing a GP's realization later on.

4.2.4 Compartment

A Compartment (CT) is a set of Entities that fulfill the following requirements:

- Each entity carries a name from a CT-specific name space (e.g., MAC addresses in the Ethernet CT). These names can be "empty" and do not need to be unique. Rules how names are assigned to entities are specific to each CT.
- All entities in a CT *can* communicate, i.e., they support a minimum set of communication primitives/protocols for information exchange. These protocols are implemented as different GP types. Hence, for joining a CT, an Entity must be able to instantiate the EP types required by the CT.
- All entities in a CT *may* communicate, i.e., there are no physical boundaries or control rules that prohibit their communication.

A special CT is the Node CT (N1 and N2 in Figure 5). It corresponds to a processing system, i.e., typically an operating system that permits communication between different processes (e.g., by using Unix domain sockets). By means of virtualization, multiple Node CTs can be created on one physical node.

An Entity is typically member of at least two CTs, the "vertical" Node CT and a "horizontal" CT. Furthermore, the Entity has a (possibly empty) set of names from each of the respective CT name spaces.

Note that GPs cannot cross CT boundaries due to the possibly different name spaces, protocols, etc. GPs always reside within a single CT.

4.2.5 Core

The Core is a special Entity. It exists once per Node CT, is only member of the Node CT, and is responsible for node-wide management. The Core mainly supports other Entities in cross-CT (i.e., cross-layer) issues, like, name resolution, mobility, managing/controlling Hooks, and service discovery. E.g., in the example in Figure 5, when setting up the GP between EP1 and EP2, it is the Core that tells E1 that E3 is able to provide the service required to realize its GP.

Note that the Core is also able to reconfigure the realization of existing, i.e., already established, GPs during runtime. This is done transparently to the involved Entities by moving a Hook from one EP to another, possibly in another CT. This feature is required, e.g., to realize mobility or to switch between different transmission modes, like cooperative and direct transmission.

4.3 GP establishment

This section describes the steps to actually set up a GP. The description is based on the scenario introduced in Figure 5.

Assume, the GP EP3 $\leftrightarrow$ EP4 in C2 is already established. Now, E1 wants to establish a GP to E2 within C1. The following steps are necessary:

1. E1 asks the Core in N1 that it intends to open a GP to E2 and that it requires the service "unreliable unicast" for this.

2. The Core (N1) knows from an earlier registration process that E3 in C2 is able to provide the requested service. Hence, the Core performs a name resolution for E2 in C2. For now, the name resolution is just a black box. Details will be given in Section 5.

3. The name resolution succeeds and shows that E2 in C1 can be reached via E4 in C2. The Core stores this result in a resolution table and notifies E1. Note that the result of the resolution is not unveiled to E1 to permit to transparently change the GP realization later on.

4. E1 instantiates EP1 and requests a Hook according to the previous name resolution from the Core (N1). The Core creates a Hook between EP1/EP3.

5. EP1 sends an OpenGP packet via the Hook. This packet contains its final destination E2 in C1 and first arrives at E3 where it is sent via the GP EP3 $\leftrightarrow$ EP4. When it arrives at E4, E4 tells the Core (N2) that an OpenGP packet has been received for E3 in C1.

6. The Core (N2) notifies E2 about the OpenGP packet. E2 in turn instantiates EP2. Thereafter, the Core (N2) creates a new Hook between EP2 and EP4 such that the OpenGP packet can be sent via the Hook to E2.

7. Finally, EP2 acknowledges the OpenGP packet to EP1.

The same mechanisms can also be used for setting up GPs in more complex scenarios, spanning more than two Node CTs and multiple horizontal CTs.

4.4 Testbed implementation

To be able to use Entity, EP, and Core implementations in various environments, like Linux, Windows, and embedded systems, we used C++ for efficiency and strictly separated the implementation of the logic parts from the environment-specific parts. In detail, this separation means that our testbed abstracts execution environment functions, e.g., by mapping Hooks to available IPC mechanisms. Entities, EPs, and Cores just use these abstractions, which enables to use their content (transport protocols, routing strategies, mobility schemes, data encoding, etc.) in different environments without changing the code.

We realized this separation by inheritance, provided by the object-oriented programming paradigm. For this, we implemented all abstractions, like Hook, timeout, and callback handling, in a root class, called `AbstractEntity` (and analogously for EPs and Cores). From this root class, wrapper classes like `PosixEntity` and `OmnetEntity` inherit to map the abstractions to the appropriate execution environment APIs. Thereafter, an `Entity` class is derived from *one* of these wrappers (chosen during compile time). Own, environment-independent Entity classes inherit from this class. Environment-specific entities directly inherit from a wrapper class. Figure 6 illustrates this.

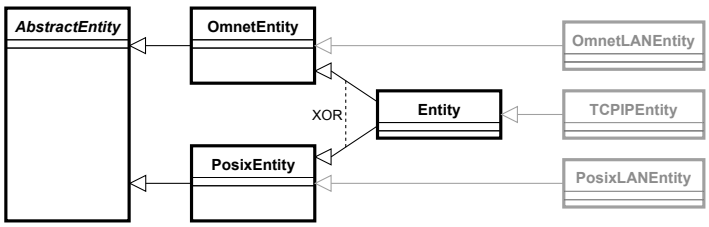


Figure 6: Inheritance graph for Entity implementations. `Entity` is usually the base class from which user-specific Entities (gray) are derived.

Currently, we have ready-to-use wrappers for POSIX-compliant systems, like Linux, BSD, and Darwin (Mac OS X) and for OMNeT++ [15], an open-source discrete event simulator. The wrapper for OMNeT++ is especially useful during the development phase and for demonstrating scalability while at the same time using the implementation in real-world scenarios via the POSIX wrapper. Additionally, we are currently working on wrappers for Windows and OpenFlow [16].

4.5 Related work

For prototyping data transport schemes, there are several existing approaches to modify today's network stack, like the Click Modular Router [17], BSD Netgraph [18], OpenFlow [16], and XORP [19]. XORP is intended for implementing new routing protocols and focuses only on IP routing and route calculation. It is based on the Click

Modular Router. Click is a software framework for Linux to construct arbitrary packet processors. However, as we needed a full architecture that provides core services, like full reconfigurability during runtime, name resolution, and path setup among multiple nodes, Click's features were not sufficient.

A very similar approach to Click is BSD Netgraph. Netgraph supplements the network stack with nodes that process packets. The disadvantages are similar to those of Click, plus that Netgraph is only available on FreeBSD.

A different approach is OpenFlow. OpenFlow is an interface to modify flow tables (i.e., packet forwarding behavior) of infrastructure nodes, like switches, routers, access points, and Linux nodes. As OpenFlow is basically "just" an interface definition, it cannot provide the required means for our purposes. However, we intend to extend our framework implementation to support OpenFlow to provide powerful processing capabilities on carrier-grade infrastructure nodes.

5 Name resolution framework

Resolving "names" into "addresses" is used in everyday networking at various layers and abstraction levels. It is realized by a patchwork of individual techniques and concepts. In current networking practice, there is no clear consensus on what a "name" or an "address" really is and how they should be mapped to each other between different layers of a system. Closely linked to this confusion is a lack of a clear concept how the entities inside the individual layers in a system refer to each other and what is necessary to identify the mapping between such two entities; only patchwork solutions for typical combinations of layers (e.g., ARP, DNS) exist. These issues make it difficult to introduce a new protocol or a new layer, as this likely to break existing name resolution schemes.

We propose a flexible yet unified name resolution framework that has two advantages: (1) With a very small set of primitives, a vast range of resolution mechanisms, like ARP or DNS, can be captured. (2) Introducing new layers is much easier. We will discuss information-centric networking as an example.

In the following, we use the framework component definitions introduced in Section 4.2 to describe our name resolution framework.

5.1 Resolution process

When resolving a name, an Entity knows the name of its desired peer Entity and the CT to which itself and this name belongs. The objective of name resolution is to find the following additional information for such a name:

- The name of a CT via which the peer Entity can be reached (e.g., "WLAN").
- An Entity inside this "lower CT", which can handle the communication on behalf of the originator Entity (typically, by means of sharing a Node CT).

- The name of a remote Entity in the lower CT that can pass on data to the actual peer Entity (typically, by means of sharing a Node CT).

The core point in designing a unified name resolution system is to avoid spreading knowledge of how to interpret a name outside of its CT. Neither does the upper CT understand names of the lower CT nor vice versa. Hence, the only thing an Entity can do to resolve a name (in absence of further knowledge) is to contact all other Entities in its own CT and ask which Entity *has* this name (optimizations will come later) – a WhoHas message is sent *inside* its own CT.

Horizontal CTs usually do not have direct communication means, i.e., spreading a WhoHas message requires assistance of suitable lower CTs (which CTs are suitable depends on the required communication service). A lower CT Entity, however, needs to be told where to send this message; it needs a remote address, which has to be provided by the higher CT Entity that initiates the resolution. Note that this lower CT remote address is, from the perspective of the higher CT Entity, a configuration parameter (opaque string) which it needs to provide but not to understand. Hence, the resolving Entity sends its WhoHas message to all local Entities in all suitable CTs, with the remote address (in the *lower* CT) being looked up, e.g., from a configuration file. This address could be a unicast, broadcast, or even anycast address inside the lower CT.

In the lower CT, the WhoHas message is distributed to its remote address, possibly to many Entities in this CT. The receiving lower-level Entity will receive the WhoHas message and will distribute this message to *all* entities in the original CT. It does *not* have to process names of the original CT.

Inside the original CT, Entities understand names contained in the WhoHas message. Each Entity checks whether it matches the desired name (it does not have to *be* the named Entity, cp. ARP). If no, it can silently discard the message. If yes, it answers with an IHave message, containing (1) the original CT name, (2) the name to be resolved in the original CT, (3) the lower CT name, and (4) the lower CT address over which the name in the original CT can be reached.

5.2 Examples

The following examples show that two very different name resolution schemes can easily be cast into this framework.

5.2.1 ARP

For the ARP example, we use the scenario in Figure 7. There are three Node CTs, two LAN CTs, and one IP CT. We want to resolve IP name 5.6.7.8 from the IP Entity with name 1.2.3.4, which needs to learn that it can reach 5.6.7.8 via the local LAN Entity 45:67:89 in CT LAN_A by sending to 89:AB:CD.

According to our framework, name resolution proceeds as follows:

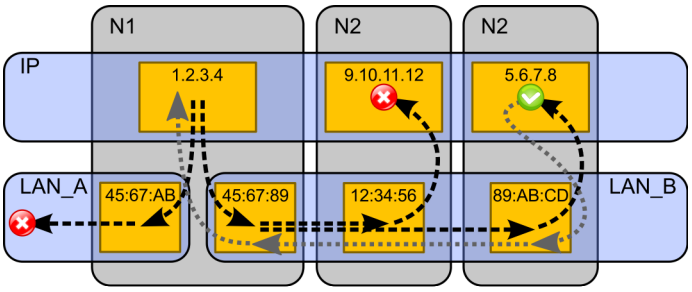


Figure 7: Name resolution in an ARP scenario. Paths of WhoHas messages are illustrated with dashed, black arrows; IHave messages with gray, dotted arrows.

1. Entity 1.2.3.4 checks (e.g., from configuration file, Core) via which CTs in can resolve IP names and finds two LAN CTs. It prepares two WhoHas packets, including: (1) IP as CT name, (2) IP originator 1.2.3.4, (3) IP destination 5.6.7.8, and (4) remote destination Entity address (e.g., from configuration file, Core), which would typically specify 00:00:00 for LAN broadcast.

2. LAN Entities distribute the packets to the destination (broadcast) address.

3. Receiving LAN Entities distribute the packet to all attached IP entities.

4. Receiving IP Entities compare the destination IP name against their own (or those names for which they know a route). IP Entity 5.6.7.8 knows that it has this name and sends back an IHave packet. It has essentially the same information as the WhoHas packet (with lower CT broadcast address replaced by LAN 45:67:89 as destination and LAN 89:AB:CD as sender).

5. The originator IP Entity 1.2.3.4 receives the IHave packet and can enter the contained information (lower CT addresses) to its resolution table (without having to understand semantics of these addresses).

5.2.2 Peer-to-peer-based resolution

Now, the resolution process is more complex, e.g., using peer-to-peer techniques. A typical application example for this could be an information-centric network as described in Section 2.

Assume a browser would like to resolve a name in the HTTP CT to which it belongs. For this, it sends a WhoHas message to a resolver. The resolver, instead of simply looking up the name in a list, would communicate inside a Distributed Hash Table (DHT) to find the destination address (in the desired underlying IP CT). For this to work out correctly, the resolver is member of two CTs: the HTTP CT as well as a DHT CT for the actual name resolution.

Figure 8 illustrates such a setup. Note that the WhoHas message in the HTTP CT is *not* simply relayed as a WhoHas message in the DHT CT; it turns into an in-CT, query-like message (a WhoHas message in the DHT CT may exist as well but serves another purpose, namely to find the IP address for a member of the DHT only known by its name in the DHT CT).

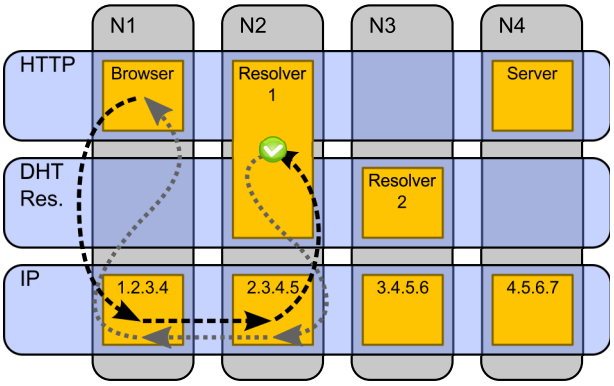


Figure 8: Name resolution in a P2P scenario.

5.3 Testbed implementation

We implemented the name resolution framework directly in conjunction with the data transport framework (Section 4.4). In consequence, the transport testbed is able to deal with any naming scheme and any name resolution implementation.

Our name resolution framework implementation spans the Core and Entities. When an Entity wants to resolve a name, it creates a WhoHas message, containing the following information: (1) its own name, (2) the name to be resolved, and (3) the CT name to which these two names belong. Thereafter, this message is passed to the Core, along with a descriptor that specifies which service will be required for data transfer later on (after a successful name resolution). The Core uses this descriptor to determine for which of the available (horizontal) CTs a resolution is performed. For each of these suitable CTs, the Core checks if an entry exists in the `ResolvConf`. An example is illustrated in Table 1; it contains two entries for configuring name resolution within the CTs `TCP_IP` and `NetInf`. Resolving a `TCP_IP` name in the `LAN_B` CT also takes place in the `LAN_B` CT; resolving from `NetInf` to `NetInfTransport` is done in the `NetInfRes` CT.

The Core extends the WhoHas message with the information from the ResolvConf and sends it to all Entities in the own Node CT that are member of the CT defined by ResolverCT. These Entities send the WhoHas message to the resolver Entity (ResolverName) where it is processed and answered.

When the resulting IHave message arrives back at the originating Entity that sent the WhoHas in the CT defined by ResolverCT, it is passed to the Core. The Core adds

Table 1: Example `ResolvConf`. SrcCT is the CT from which a name is resolved, DstCT the CT to which the resulting address belongs, ResolverCT the CT in which resolution takes place, and Resolver the name of the resolver Entity.

SrcCT	DstCT	ResolverCT	Resolver
`TCP_IP`	`LAN_A`	`LAN_A`	`00:00:00:00:00:00`
`TCP_IP`	`LAN_B`	`LAN_B`	`00:00:00:00:00:00`
`NetInf`	`NetInfTransport`	`NetInfRes`	`12345`

the contained information to its `ResolutionTable`. An example is shown in Table 2. It holds two results; one for 5.6.7.8 in the `TCP_IP` CT that succeeded in the `LAN_B` CT and one resolution that was answered in `LAN_A`.

Table 2: Example `ResolutionTable`. SrcEntity is the Entity that starts the resolution, DstEntity the name to be resolved (both within CT). LowerSrcEntity/LowerDstEntity are the Entity names in LowerCT that were found during resolution.

CT	SrcEntity	DstEntity	LowerCT	LowerSrcEntity	LowerDstEntity
`TCP_IP`	`1.2.3.4`	`5.6.7.8`	`LAN_B`	`45:67:89`	`89:AB:CD`
`TCP_IP`	`1.2.3.4`	`2.3.4.5`	`LAN_A`	`45:67:AB`	`23:45:67`

The Core now informs the Entity that requested the resolution (1.2.3.4 in the example) about the success and returns a reference to the `ResolutionTable` entry. The Entity cannot read the `ResolutionTable` content. It uses the pointer to reference the name resolution result when sending data to the resolved destination name. This strictly keeps names in their own CTs and makes common misuse impossible (e.g., to put "lower addresses" like IP into the payload).

5.4 Related work

To our knowledge, this is the first approach towards a unified name resolution framework that captures a wide range of existing and future resolution services and spans over all technological levels, i.e., network layers.

6 Conclusion

We presented four frameworks for prototyping future Internet techniques related to data transport, information-centric networking, naming, and name resolution. These

frameworks can be used on their own to prototype individual concepts or can be combined to complex testbed implementations, like we did it for the integrated NetInf prototype.

Experience during our work confirmed that the frameworks simplify prototyping a lot. Due to their generality, they will also be useful for other projects as they reduce redundant implementation of basic testbed functions and provide ready-to-use building blocks to rapidly complement new, small components with an overall network architecture. We will publish our code as open-source project.

Acknowledgments

We gratefully thank the project groups AugNet I and II for valuable discussions and their excellent implementation support. Furthermore, we would like to thank our colleagues in the 4WARD project for their input and fruitful discussions.

References

[1] 4WARD Consort., "4WARD – Architecture and design for the future Internet," 2008. [Online]. Available: http://www.4ward-project.eu/

[2] B. Ahlgren, M. D'Ambrosio, C. Dannewitz, M. Marchisio, I. Marsh, B. Ohlman, K. Pentikousis, R. Rembarz, O. Strandberg, and V. Vercellone, "Design considerations for a network of information," in *Proc. ReArch2008*, Dec. 2008.

[3] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. L. Braynard, "Networking named content," in *Proc. ACM CoNEXT*, Dec. 2009.

[4] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *Proc. ACM SIGCOMM '07*. New York, NY, USA: ACM Press, 2007, pp. 181–192.

[5] OneLab Consort., "OneLab – future Internet test beds," 2006.

[6] A. Keller, T. Hossmann, M. May, G. Bouabene, C. Jelger, and C. Tschudin, "A system architecture for evolving protocol stacks," in *Proc. ICCCN*, Aug. 2008.

[7] G. Tyson, A. Mauthe, T. Plagemann, and Y. El-khatib, "Juno: Reconfigurable middleware for heterogeneous content networking," in *Proc. NGNM*, Sept. 2008.

[8] C. Dannewitz and T. Biermann, "Prototyping a network of information," Prototype demonstration at the IEEE LCN, Oct. 2009.

[9] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proc. ACM SIGCOMM '03*. New York, NY, USA: ACM Press, 2003, pp. 163–174.

[10] Google, "Guice," Mar. 2007, open source project. [Online]. Available: http://code.google.com/p/google-guice/

[11] ——, "Google protocol buffers – Protobuf," July 2008, open source project. [Online]. Available: http://code.google.com/p/protobuf/

[12] T. Biermann *et al.*, "D-5.2.0: Description of Generic Path mechanism," Jan. 2009, project deliverable.

[13] P. Bertin, R. L. Aguiar, M. Folke, P. Schefczik, and X. Zhang, "Paths to mobility support in the future Internet," in *Proc. IST Mobile Comm. Summit*, June 2009.

[14] T. Biermann, Z. A. Polgar, and H. Karl, "Cooperation and coding framework," in *Proc. IEEE Future-Net*, June 2009.

[15] A. Varga *et al.*, "OMNeT++ discrete event simulation system," open source project. [Online]. Available: http://www.omnetpp.org/

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[18] A. Cobbs, "All about netgraph," Mar. 2000, daemon News.

[19] M. Handley, O. Hodson, and E. Kohler, "XORP: An open platform for network research," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 53–57, 2003.