Karlsruhe Institute of Technology (KIT)
Institute of Telematics

TELEMATICS TECHNICAL REPORTS

Evaluating a Framework for Different Networking Paradigms

Denis Martin, Hans Wippel
{martin, wippel}@kit.edu

July 10, 2013

TM-2013-2

ISSN 1613-849X

http://doc.tm.kit.edu/tr/

Institute of Telematics
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany

Evaluating a Framework for Different Networking Paradigms

Denis Martin, Hans Wippel
Institute of Telematics, Karlsruhe Institute of Technology (KIT)
{martin, wippel}@kit.edu

***Abstract*—While we can observe a fast evolution and innovation of link-layer technologies and networked applications, the core Internet architecture with TCP/IP remains rigid. New protocols and networking paradigms such as content-centric networking exist but suffer from global deployment issues, acceptance, and use by application developers. In order to address these problems, we previously proposed NENA, a framework that aims at a better decoupling of applications and networks. In this paper, we evaluate the framework's concepts and interfaces to determine the minimum set of invariants needed to operate protocol families that differ in their basic abstractions and paradigms. As a basis, we used adaptations of prominent protocols, both existing and new approaches: TCP, CCNx, BitTorrent, the Bundle-DTN protocol, an MQTT message broker, and the extensible IP-replacement XIA. We demonstrated that it is possible to realize a framework for multiple diverse protocols and paradigms while introducing only a small set of invariants.**

I. Introduction

Although we observe fast evolution and deployment cycles below IP and above the socket API, the core network protocol stack remains rigid. Changes to existing as well as deployment of new networking protocols are slow at best, as can be observed with the introduction of IPv6 or SCTP. Big providers, however, see the need to make changes to currently deployed protocols and to tailor them to their needs. Google, for instance, touches core protocols that weren't changed much during the past decades (e. g. with SPDY and TCP Fast-Open) and experiments with new transport protocols such as QUIC, which recently appeared in Google's web browser Chrome. Moreover, content providers could benefit from completely new networking paradigms such as Content-Centric Networking (CCN). CCN promises better resource utilization for content distribution than traditional host-to-host or end-to-end communications. However, such solutions may not be optimal for every use case. For example, interactive communications such as VoIP are inefficient in CCN approaches.

We therefore have reason to believe that there is no "one-size fits all" solution that solves all challenges and provider needs equally well. Instead, for every challenge or use case, the most appropriate solution should be used. Thus, we envision an Internet scenario in which nodes do not only connect to a single, general-purpose network. Instead, nodes connect to different networks, each tailored to a specific use case and the needs of the content provider. Examples for such networks are online-banking networks (optimized for security), video streaming networks (optimized for real-time transfers to many users), content distribution networks (optimized for efficient data distribution), and online gaming networks (optimized for low latency). Ultimately, this enables innovation, evolution, and competition not only at link and application layers, but also in-between, at the core of the network protocol stack.

To realize such a multi-network approach, a generic framework is required to allow end-systems to easily connect to different networks and to allow easy deployment of networks. In addition, the networking API used by application developers needs to hide networking specifics for a better decoupling of application and network technologies. NENA [1] is our proposal for such a framework. NENA runs on every participating node and provides the following features: (1) concurrent operation of multiple protocol stacks, (2) component-based architecture to ease protocol composition, and (3) a unified, high-level interface for applications.

In this report (which is an extended version of [2]), we analyze and evaluate the concepts of such a multi-network framework based on our NENA prototype [3]. To this end, we implemented several different networking protocols and paradigms for NENA. Examples are (1) stream and datagram-based transport protocols with TCP/UDP-like protocols and a video streaming protocol, (2) content-based networks with a REST protocol, a CCN, and a P2P file sharing protocol, and (3) protocol-agnostic network services with a DTN implementation and a Pub/Sub-oriented message broker.

The results of this architectural evaluation reveal the minimum explicit and implicit invariants that are necessary for a framework for different networking paradigms. Those invariants describe mandatory interfaces from which the application-to-network interface is the most crucial one. With those invariants, we provide indications for the design of similar APIs and frameworks that aim at overcoming the rigidness of the TCP/IP stack.

The remainder of this paper is structured as follows: In Section II, we review selected related work that aim at raising abstractions at the API level and/or at increasing flexibility within the network stack. After that, we give a brief overview of NENA and its API in Section III. In Section IV, we describe our evaluation approach, which is subsequently used for a per-use-case analysis in Section V and an overall evaluation in Section VI. Finally, we conclude with Section VII.

II. Related Work

Related work can be divided into two areas: (1) proposals for changes of the socket API (the major invariant regarding

network protocol access by applications), and (2) proposals for extensible base architectures for the Internet (replacing IP).

The need for shifting the API to a more abstract level is recognized, for instance, by [4]: Here, an extended HTTP is proposed as the new high-level general-purpose protocol to be used by any application. Although it lacks the definition of an API, HTTP commands can be mapped to function calls. One drawback of this approach is that a web-server is always required and no peer-to-peer communication is provided. However, the main conclusion of the authors was that HTTP commands are sufficient for any use case if extended by datagram support. Activity within the IETF (name-based sockets, NBS) also aims at redesigning the socket API to achieve a better decoupling of applications and network protocols [5]. While name-to-address resolution, service names, and transport selection based on service names are considered, NBS is limited to port numbers for services and to UDP, TCP, DCCP, and SCTP for transports. In [6], the NBS approach is generalized: General transport services such as reliable transport or in-order delivery were extracted in analyzing UDP, TCP, SCTP, and DCCP. The current socket interface was then extended to make those general transport services configurable via socket options, for instance. While this API hides the actual transport protocol used, it still requires manual name-to-address resolution and application service selection.

Those API considerations are mostly based on current protocols. With ChoiceNet [7], XIA [8], and FII [9], approaches exist that aim at fostering evolution and innovation in allowing different protocols (or protocol stacks) within a domain or network. ChoiceNet aims at encouraging alternatives at different layers of the protocol stack. To increase competition and incentives to build such alternatives, ChoiceNet proposes an economic model that also takes transit providers into account. This economic control plane is the main focus of ChoiceNet. The eXpressive Internet Architecture (XIA) aims at replacing IP with a new protocol family similar to IP. The major difference to IP, however, is, that XIA supports different and extensible communication *principals*, e. g. host-based or content-based. XIA also defines a new socket API and a new addressing scheme via directed acyclic graphs (DAGs) that allow fallback addressing in the case new principals are not supported by transit domains. The support for different communication principals makes XIA an ideal candidate for a closer analysis which we describe in Section V-H.

Contrary to XIA, the Framework for Internet Innovation (FII) does not define protocols for the data plane but mostly concentrates on the control plane. FII describes three core interfaces: an interdomain interface (for interdomain routing), an interface between applications and network, and an interface for DDoS protection. For interdomain routing, FII defines the concept of *pathlets* which essentially are (virtual) path segments that can be announced by domains. Pathlets can specify their own packet format, but for an end-to-end path, only compatible pathlets can be combined. The application interface (netAPI) described by FII allows for extensibility by defining *schemas* an application can query (e. g., traditional sockets, Pub/Sub, RPC). This way, the API can be extended with additional semantics. FII describes additional mandatory interfaces to cope with architectural heterogeneity: for meta-negotiation (to negotiate a negotiation protocol), for triggering bootstrapping (i. e. attachment to a network), and for API schema queries. In total, a lot of smaller invariants are defined that way, which we think is not necessary (see Section VI-D).

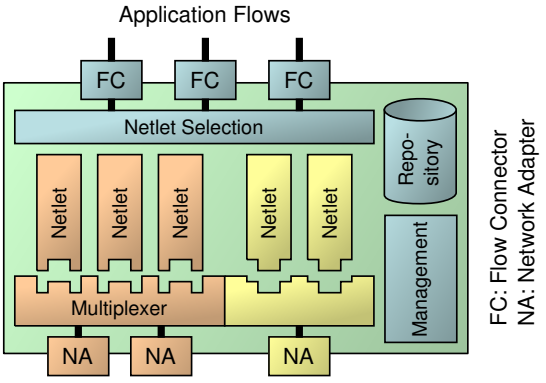

Fig. 1. Netlet-based Node Architecture (NENA)

III. Framework and API

In this section, we present an overview of our framework. A detailed description can be found in [1].

A. Network Attachment

In the multi-network scenario described in Section I, client nodes dynamically establish virtual links to multiple (virtual) networks where each is tailored to the needs of the respective content provider. In this scenario, the network attachment and protocol retrieval processes need to be automated. When an application requests a communication service that cannot be served with currently available network attachments and protocols, a lookup at a global registry is performed. The resulting information contains the necessary protocols, pointers to repositories where to download those protocols from, and the points of attachments for the requested network. The network attachment could be easily realized over the current Internet: the lookup is done using DNS service records, the repositories are web servers, the points of attachments are gateways for the provider's network, and the virtual links to the network are established using tunnels. However, with respective support within the network infrastructure, the networks and virtual links could also be realized by using virtualization techniques, which offer better support for QoS. Details of the attachment process can be found in [10].

B. NENA

The Netlet-based Node Architecture (NENA, Figure 1) is a runtime framework that allows nodes to simultaneously connect to multiple networks at the same time. These networks can be based on different protocol families using specialized network protocols. Applications access those protocols via an API that abstracts from networking details: Instead of providing network addresses and protocols, applications specify globally unique *names* as URIs and *requirements* on the

requested communication service to initiate communication with the content, service, or host associated with the name.

Protocols or complete protocol stacks in NENA are encapsulated in so-called *Netlets*. Netlets are composed of protocol *building blocks* whereas each building block contains either a complete protocol or a single protocol mechanism. The Netlet Selection component performs the selection of networks and Netlets using the requested name and the application requirements as selection criteria. This is a two-step process, where first each protocol family is queried to deliver a set of Netlet candidates that are able to fulfill the request. This can be seen as a filtering process that takes the URI, API primitive, and the requirements of the request into account. Second, the "best" candidate is chosen (e. g., the one providing the best QoS if such information is available).

Below the Netlet Selection component, the components of each currently active network are located. Multiple networks with individual protocol families can be in use on a node at the same time. A protocol family instance consists of a set of Netlets, a multiplexer, and a set of network accesses. A multiplexer constitutes the base-layer of a protocol family, performs Netlet multiplexing, and – depending on the family – implements addressing and forwarding mechanisms. A protocol family that, for instance, realizes a late-binding of names to addresses, may do name-to-address translation and network access selection here. Network accesses represent a physical or logical network interface card. Thus, multiplexing of multiple virtual networks over the same physical network has to be realized outside of NENA (e. g. via network virtualization).

For management and maintenance purposes, NENA also has network independent repository and management components. The repository component loads and instantiates Netlets and multiplexers from external sources (cf. Section III-A). The management component is the central entry point for management requests, which means that it propagates requests to the respective protocol families and collects monitoring information from different levels (node-level, protocol-family-level, or protocol-level) [11].

C. API

The main goal of the Internet scenario described at the beginning of the paper is to allow independent evolution of both applications and networks. However, the networking API will still be one of the most important invariants in this scenario. Thus, the API should hide networking details such as address formats and protocols. Furthermore, the API is required to be generic enough to cope with different communication paradigms like host-based or content-centric communications, and it needs to be flexible with respect to the introduction of new networks and protocols. Neither should require change in existing applications. In order to achieve these goals, we use globally unique names and moved name resolution and protocol selection below the API.

Based on [12], we implemented an API fulfilling the beforementioned goals. Its usage pattern is similar to today's socket API: a communication end-point is created with a primitive

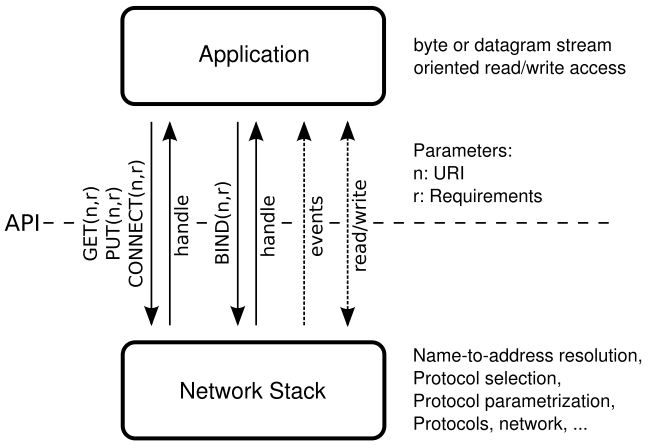

Fig. 2. API Overview.

that returns a handle on which read/write operations may be performed. This basic pattern has proven to be very portable. Any high-level abstractions such as callback-based interfaces for event-driven applications can be realized with the target programming language's means (as it is done today with the socket API).

Instead of providing only one primitive to create communication end-points, we provide five (Figure 2): CONNECT, GET, PUT, BIND, and ACCEPT. While CONNECT matches today's socket call semantics closest, GET and PUT are introduced in order to support the many contemporary applications that use a RESTful interface (i. e., HTTP) as their basic communication abstraction [4]. By providing those methods at the API level, network services do not need to use HTTP as their basic communication abstraction, and eventually new communication paradigms such as CCN can be easily introduced.

An application providing a service or content as a server uses BIND in order to announce its availability to serve requests. Instead of port numbers to identify services, the application specifies an URI. Here, wild-cards and longest-prefix matching allows an application to register itself for a base-URI. Requests with URIs containing this base-URI are sent to the application. This way, file-server or web-server applications can be easily created without requiring the application to implement protocols such as FTP or HTTP. Upon an incoming request, the serving application calls ACCEPT in order to create a new handle. From this handle, the application can retrieve meta information such as the request method (GET, PUT, CONNECT), the remote end-point's URI, and the requested properties (e.g., content-types understood by the client application). Generally, CONNECT represents byte-stream-oriented end-points, while GET and PUT allow datagram-based end-points with application data unit lengths.

In our prototype, requirements are exchanged as JSON encoded objects from the application to NENA. A simple example for a requirement JSON object is `{ "content-type": "image/jpeg", "reliable": 1 }`. This provides flexibility as well as extensibility and allows protocol families and applications to introduce new requirements.

C_1	Basic communication service usable?
C_2	Advanced options/services usable?
C_3	Naming through URIs suitable?
C_4	Services transparent to applications?

TABLE I
API EVALUATION CRITERIA

IV. EVALUATION METHOD AND CRITERIA

Unfortunately, architecture and concept evaluations do not deliver solid numbers that are easy to compare. We therefore evaluate our framework based on a methodology similar to [13]: We perform an analysis of the invariants imposed on different protocol families by our framework. Invariants in this context are architectural anchors that cannot be changed. Invariants are therefore both necessary (i. e. the key purpose of an architecture) and a burden (since they limit flexibility). The design of an architecture can be seen as a definition of explicit invariants. However, some design choices result in additional *implicit* invariants that were not necessarily foreseen by the designers. Those implicit invariants can only be revealed by a thorough analysis of the architecture and by means of use cases.

With respect to our framework, this means that we need to apply different protocol families to our framework that ideally are very diverse in their intentions and purpose. We did this by implementing existing and new protocols in a prototype implemention of our framework. The conclusions we draw from that should reveal (1) implicit invariants and (2) unnecessary explicit invariants of our framework. The result will then be the minimum set of architectural invariants necessary for a network framework for a diversity of different networking paradigms. Although it is impossible to say whether this set is comprehensive (since we cannot foresee future networking paradigms), it should cover a broad range of currently discussed approaches.

The main evaluation criterion is the API itself. To analyze this major invariant, we define some criteria for different aspects of the API in Table I: The API was designed as a single replacement for the current socket-API. The major semantic additions to the stream and datagram services are additional primitives and the use of requirements. Criteria C_1 and C_2 cover whether those primitives are sufficient to exploit the full potential of the evaluated communication services. As communication end-points, URIs are used instead of addresses and port numbers. In addition, the meaning of an URI is not limited to a service at the application-layer, but could also be directly addressed content. Whether URIs are flexible enough, is covered by evaluation criterion C_3. Different communication paradigms may result in different expectations of the users (applications) of those paradigms. To what degree this can be hidden by a generic API is covered with criterion C_4.

V. ANALYSIS AND EVALUATION PER USE CASE

In this section we briefly describe the protocols and protocol families we used for our evaluation. In addition, we describe their particularities especially w. r. t. to our API and highlight respective findings for our evaluation. Their realizations within NENA are summarized in Figure 3.

A. Current Protocols

Starting point of our evaluation is how well current communications abstractions fit into our framework. At the lowest abstraction, an application may request a traditional socket where it also selects the transport protocol that is to be used. This can be done with a canonical mapping from the URI to a traditional socket, e. g., `udp://example.com:4200`. Raising the abstractions a bit, we can use the first parts of an URL: `http://example.com` is expected to create an end-point for the application, on which it can expect HTTP formated messages (i. e., the application implements HTTP by itself). The framework has to map the service (http) to a protocol (e. g., TCP) and protocol parameters (e. g., port 80). This mapping can be static or additional resolution mechanisms can be triggered (e. g., by querying the DNS SRV records in order to resolve the transport protocol and port number). The resolution of IP addresses and the decision whether IPv4 or IPv6 is used is also left to the framework. The application is not involved in IP address or protocol selection: If a web server provider decides to deploy his service via SCTP/IPv6, client nodes can automatically use this protocol combination regardless of the application triggering the request. Thus, already in this scenario, the name-based approach yields obvious advantages. Note that this essentially describes what was envisioned by NBS (cf. Section II). However, the URIs and the addition of application requirements increase the extensibility of our approach as will be shown with the following examples.

B. Advanced Transport Protocols

An example we often use for demos is video transport. With the URI namespace (`video://`), a lot of semantics is already defined: the information exchanged will be video data intended for live streaming. The format of the data is predefined by the namespace and can be extended with additional application requirements, e. g. `content-type`. This knowledge allows for sophisticated transport adaptations if necessary. If, for instance, the packet loss experienced during communication increases, the transport may add information for forward error correction to the data. Since it knows the characteristics of the data, it may apply this information only to the base video-layer in order to save bandwidth.

C. REST

Representational State Transfer (REST) is a concept that describes the transfer of resources using a stateless API between clients and servers. HTTP is the most prominent example of a RESTful API. Our implementation consists of a building block that, upon a new application request, transfers the requested URI and the used API primitive to the corresponding server. Thus, it realizes a minimum protocol that fully features the API described in Section III-C.

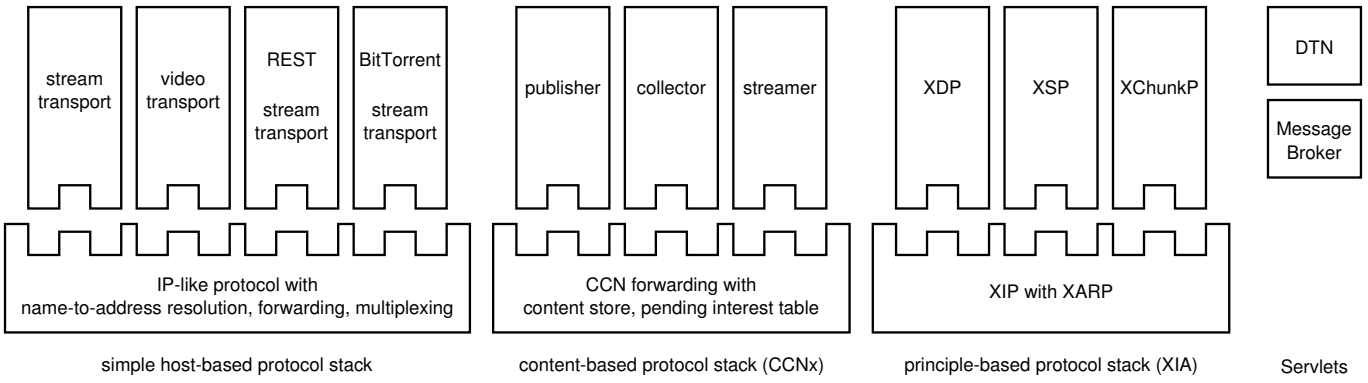

Fig. 3. Overview of the protocols and protocol families realized with NENA.

D. BitTorrent

BitTorrent is a peer-to-peer protocol that uses chunked content retrieval. This means files are not downloaded from a single peer as a continuous, sequential stream. Instead, files are partitioned into smaller chunks and these chunks are downloaded in arbitrary order from a set of nodes (a *swarm*). Meta information on content is stored in Torrent files, which are used to eventually retrieve a list of peers that either have the requested content or are also interested in retrieving it.

We decided to hide the concept of Torrent files from applications. Thus, we had to add a mapping from URIs to Torrent files, a feature that is not part of BitTorrent itself. The semantics of the API primitives are as follows: GET retrieves an object specified by the URI, and BIND is used to publish an object with the specified URI, triggering the generation of a Torrent file. PUT and CONNECT are not supported by our implementation. CONNECT, however, could be used for BitTorrent Live, an upcoming P2P streaming protocol. Retrieval of large objects with GET is a challenge. The application expects a sequential file stream from the handle, but the chunk retrieval order is not sequential. This means that potentially large amounts of data must be cached by the framework before any data can be passed to the application.

E. CCN

Content-centric networking (CCN) [14] is a communication paradigm in which content is directly addressed by its name within a network without specifying any host addresses. When content is transfered over a network, network nodes on the data path are able to cache the content. Thus, further requests can be served by such network caches, which ultimately reduces network traffic.

We choose CCNx[1] as a reference for our CCN implementation. The implementation consists of three Netlets and a Multiplexer (Fig. 3). One Netlet realizes the retrieval of content from the network (*collector*), another Netlet is used for the publication of content in the network (*publisher*). The third Netlet (*streamer*) allows running interactive communication over a CCN network [15], basically implementing retrieval, publication, and additional components for the coordination of the data stream. All Netlets operate over a common multiplexer that manages forwarding, pending interests, as well as content caches. The API primitives are used as follows: GET retrieves an object named by its URI, and PUT publishes a new object under the given URI. After all data from the application is transfered to the framework when using PUT, the data is cached locally and the application may close its API handle. CONNECT and BIND are used to create an interactive data stream over CCN. Since, however, CCN is not optimal for interactive streaming, we normally would not offer the streamer Netlet to applications. Instead, other networks with different protocol families should be used.

[1]http://www.ccnx.org

F. DTN

Delay-tolerant networking (DTN) describes communications between hosts through networks with high and variable delay or even with intermittent connectivity. It uses a store-and-forward mechanism: a message (also called *bundle*) can be stored on intermediate nodes until the next hop on the path to the destination node becomes available. One realization of such a protocol is the Bundle Protocol (BP) [16] which we took as a reference for a DTN implementation in NENA.

The BP is an application-layer protocol that is designed to operate on URIs as names and on any protocol family across different networks (not just TCP/IP). The name-based addressing and the cross-network feature made the BP an interesting candidate for our evaluation. The cross-network feature, however, showed a general interworking issue between protocol families within NENA: data received via a Netlet of one family needs to be passed to a Netlet of another family. We could have built the BP based on building blocks, replicating its functionality in multiple Netlets for different protocol families and realize some inter-Netlet communication. This, however, would have required coordination between BP instances. For this reason, we decided to extend NENA by an additional concept named *Servlets* (Figure 4): Servlets carry out services that today are located above the transport-layer. They can bind themselves to URIs (as applications can do) and react on requests from remote nodes. Optionally, they may also register themselves at the Netlet Selection component and offer their services directly to applications (e. g., as middleware). Basic communications via DTN work

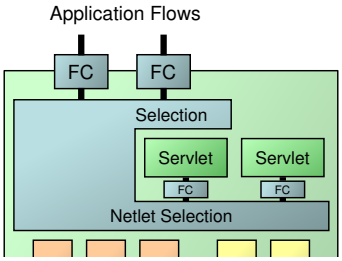

Fig. 4. NENA with Servlet extension

well with BIND and PUT on DTN-URIs. To allow applications to specify a lifetime of a bundle, the set of requirements needs to be extended by `lifetime`. However, DTN also allows for optional status reports, which may also be delivered to applications after a system restart. Although application status reports are supported, they currently are not delivered to applications after they disconnect from the framework. This would require creation of a persistent handle that can be reused when the application reconnects to the framework at a later time. Last but not least, the concept of DTN is quite different from what most applications are used to expect when using a communications API: Very long delays (e. g., with GET) may result in undesirable application behavior if the application is not aware of the DTN.

G. Pub/Sub

A common communications abstraction is the Publish / Subscribe paradigm: Instead of establishing a direct communication between applications, messages are exchanged indirectly via a topic. New messages are published to the topic and all subscribed applications are notified. One protocol realizing a message broker based approach is the Message Queue Telemetry Transport (MQTT) [17]. The message broker stores messages published to a topic and notifies all subscribers.

Since the message broker is a network service that does not need a local application, we decided to realize it also as a Servlet (see previous subsection). The broker binds itself to a wildcard URI, e. g. `broker://instance/topic/*`. Publishing applications use the PUT primitive with an URI containing the topic, e. g. `topic://instance/topic/temperature`. The MQTT Netlet on the publishing node translates this URI to the broker URI and forwards the published message. A subscribing application uses the BIND primitive on the same topic URI. The MQTT Netlet translates this to a subscribe request and forwards it to the respective broker. In addition to this Pub/Sub interface, the GET primitive is used to retrieve the latest message published to a given topic without creating a subscription.

With the API primitives BIND and PUT as used in this example, a multicast service can be utilized as well. Instead of a message broker, a multicast group identified by the topic URI is used to distribute messages. BIND triggers a group join, while PUT sends a message to the group. Whether a broker (which is able to store messages) or a multicast group (which only allows direct but efficient forwarding) is used, must be decided based on the application's requirements.

H. XIA

As already mentioned in Section II, XIA defines a complete new base architecture replacing the IP protocol family. Due to its architectural similarity to IP (it defines protocols corresponding directly to existing protocols such as ARP, UDP, TCP, DHCP etc.) its implementation design was straight forward (Figure 3, right): XIP, the base-layer protocol, is realized within the multiplexer, while the transport protocols XDP (unreliable datagram transport) and XSP (reliable stream transport) are realized in different Netlets. The XChunkP realizes a CCN-style protocol. One key advantage of XIA's integration into NENA is the simplification of the API. XIA's API is closely related to today's socket API, which means that applications need to do name resolution: for a given URI, applications have to resolve a corresponding DAG before they can open a socket. In NENA, this step is hidden from applications, and applications are not even aware of XIA.

However, the support for different principals (e. g. for host-based or content-based communications) requires application knowledge. Since with XIA's API applications are responsible for providing a DAG, the principal types are known to the application. With our API, the principal type must be part of the URI namespace definition, which is in line with the use of URIs in the previous examples.

VI. Overall Evaluation

In the previous section, we analyzed the realizations of different protocols and protocol families within NENA. In this section, we draw conclusions from this analysis and discuss the impact on the framework's API and other invariants.

A. API Usage

The API usage and the evaluation criteria as results from the previous section are summarized in Table II[2]. Regarding the primitives, not all are supported by every protocol. This was to be expected, since protocols are designed for specific purposes and the API covers a broad range of purposes. When selecting an appropriate protocol, this already allows a filtering by the requested primitive.

Based on the API evaluation criteria, we summarize the API usage as follows: All basic communication services of the protocols can be used (C_1). However, when it comes to advanced features (C_2) of the protocols, such as additional primitives of HTTP (DELETE, HEAD), those cannot be used directly and must be realized via requirements. For instance, to retrieve only meta-data of a resource (HTTP HEAD), the application has to issue a GET with a requirement such as `metadata`. For some primitives, this may not be intuitive to the application developer. In such cases, we propose to add an additional layer of abstraction *above* our API. Similar to FII's API schemas [9], this allows for a tailored API in certain cases. But instead of adding the support for API schemas as an invariant to our framework, we believe that application-layer abstractions are sufficient. Regarding naming, URIs as

[2]Although CCNx has a solution for streaming data, we did not include CONNECT/BIND since we think that this functionality is not desirable here.

API usage	Stream	Video	REST	BitTorrent	CCNx	DTN	Pub/Sub	XIA
GET(name, requirements)	—	—	✓	✓	✓	—	✓	✓
PUT(name, requirements)	✓	✓	✓	—	✓	✓	✓	✓
CONNECT(name, requirements)	✓	✓	✓	✓	—	—	—	✓
BIND(name, requirements)	✓	✓	✓	✓	—	✓	✓	✓
Criteria								
C_1 – Basic communication service usable	✓	✓	✓	✓	✓	✓	✓	✓
C_2 – Advanced options/services usable	—	—	✗	✗	✓	✗	✓	✓
C_3 – Naming through URIs suitable	✓	✓	✓	✓	✓	✓	✓	✓
C_4 – Service transparent to application	✓	✓	✓	✗	✓	✗	✓	✓

Dashes (—) mean that the API primitive is not supported, resp. that there are no advanced options.

TABLE II
EVALUATION OVERVIEW: USAGE OF API PRIMITIVES AND EVALUATION CRITERIA.

names have proven to be suitable (C_3). The main reason is that they allow an extensible way to *structure* names with their namespaces and that different *semantics* can be associated to those namespaces.

The main concern, however, is transparency of the communication service offered by the respective protocols (C_4). When using the socket API today, applications make an implicit assumption about the service underneath: direct communication is possible. When raising abstractions, implicit assumptions have to be rethought and formulated explicitly. When using DTN, for instance, the application must be aware of the fact that a transfer may take hours or days. Thus, the application should bundle all information at once and should not expect an immediate feedback about the success of the transfer. A similar issue, though not that severe, arises when using chunked transfers (e. g., BitTorrent): The API delivers data sequentially, so the protocol needs to buffer the data before it can be sent to applications. A viable solution for this is the use of delegated transfers as described in [12]: Here, the application tells the framework where the requested data should be delivered to (which in this case could be a file system resource).

B. Explicit Invariants

In this section, we review the implications of our design choices – i. e., our explicit invariants – based on the experiences with a diverse set of protocols.

URIs as names. In some situations, applications do not need to know anything about the name structure of the object they are requesting, so the name is just seen as a flat string of characters. However, when it comes to application expectations, it makes a difference, if the name refers to a content object or a service. We therefore think that URI namespaces play an essential role for a name-based API, and that these need to be standardized. With this standardization, the structure, the semantics, and the set of requirements of the names are defined.

API primitives. The basic set of API primitives we use cover a large part of communication *intents*. In some cases, additions to those primitives make sense in order to build higher abstractions and simplify the life of application developers (such as an API that directly offers the primitives Publish and Subscribe). However, we don't think that it is necessary to include those in the basic API primitives. Instead, higher abstractions can be built on top of the basic primitives.

Application requirements. On one hand, application requirements are used as the major extension point of the API. On the other hand, a basic set of requirements must be standardized to create a common understanding between applications and protocol stacks. Combined with the standardization of URI namespaces, this yields a powerful extension point where new namespaces can be accompanied with a new set of requirements.

Query interface. When an application invokes a communication request, the framework needs to query the available protocol stacks whether they are able to handle the request. In this query, the name, primitive, and requirements are passed to the protocol stacks, and those need to respond with a list of protocol "candidates" to serve the communication request. For this query, an interface needs to be implemented by each protocol stack.

Netlets, Multiplexers, and Servlets. The candidates mentioned above are components that need a common interface to pass application and network data to. In our case, these components are Netlets, and its interfaces need to be implemented by each protocol that is encapsulated in such a Netlet. A Multiplexer represents a base layer for a protocol stack. However, we did not identify any relevant difference between a Netlet and a Multiplexer concerning its handling by the framework: both are external components retrieved and loaded on demand. We therefore conclude that the differentiation between Netlets and Multiplexers is an unnecessary explicit invariant. However, it defines some structure that aids the protocol stack designer to distribute functionality. Servlets were added as additional networking components to allow middleware-like protocols within NENA. But as with the Multiplexer, we did not identify any further advantage other than giving the protocol stack designer a way to structure his protocol design.

Network attachment. If no candidates for a given communication request are found, a network lookup, protocol retrieval, and network attachment process need to be performed. However, we do not need to define common protocols for such actions, since those can be realized as well with new protocol families. The easiest way to start deployment would be to use the current Internet as a base network for these actions (see Section III-A).

C. Implicit Invariants

Based on the experiences gained when integrating different protocols and protocol families, we discovered some additional implicit invariants. While some of them can be seen as implementation issues, they are still of practical relevance.

Flow control. Application flow control mechanisms are part of the respective transport protocols. However, the concrete mechanism to control the rate at which applications send their data to the framework needs to be realized by the framework itself. Thus, there needs to be a flow control interface between protocols and the framework.

Flows. When receiving incoming data, protocols must be able to identify the corresponding local application. This requires a handle or ID for the respective application, or a reference to the respective flow connector. This ID or reference only has local validity and must not be used by protocols outside the host-local context (which is done today with port numbers). Thus, this flow ID is no invariant by itself, but protocols need to be careful not to add additional semantics to it. Instead, flow identification outside the host-local context should be based on URIs or protocol specific IDs.

User feedback. In some circumstances, a user callback is necessary, e. g., for authentication or network attachments. This can be realized either through API events (which has the disadvantage that applications need to implement handlers for them) or through operating system requests.

Session management. User sessions on web-sites, for instance, are quite common today. Today, a combination of client-side states (cookies) and server-side states (hold in a data store) are used. While such session information can be exchanged via requirements, additional standardization is necessary here to describe the exchange format.

D. Non-Invariants

Regarding the explicit invariants, we define fewer than XIA or FII (see Section II). The main reason for this is, that we do not consider interworking between networks of different protocol families – which we believe is not necessary: Virtualized networks can be administered globally by a single provider, so no horizontal interworking is necessary here. Infrastructure providers (which roughly can be seen as transit providers for those virtual networks) still need horizontal interworking between domains. This, however, can be solved on an individual basis between providers (or between virtualization technologies) and does not require global agreement.

VII. Conclusion

In this paper, we outlined how we realized different existing and upcoming protocols with NENA and critically analyzed the invariants of our proposal. We showed that it is possible to realize a framework for multiple diverse protocols and paradigms while only introducing a small set of invariants. From those invariants, the API proved to be the most important one. The name and requirement-based API offers the necessary flexibility for supporting a diverse set of protocols. However, the evaluation showed that advanced communication services and special use cases need application awareness: In cases such as DTN, today's socket-based expectations on the communication service (e.g., regarding delay or service availability) may not hold. Thus, we propose to put more semantic in names and to standardize URI namespaces as well as namespace specific requirements. Contrary to existing approaches, we conclude that further invariants for network interoperability are not necessary. Although network interoperability is, of course, required for interworking domains (such as the current Internet), it is not necessary if provider-specific networks are used instead of a single global Internet. This can be achieved with virtualization or tunneling techniques.

References

[1] D. Martin, L. Völker, and M. Zitterbart, "A Flexible Framework for Future Internet Design, Assessment, and Operation", *Computer Networks*, vol. 55, no. 4, pp. 910–918, Mar. 2011.

[2] D. Martin and H. Wippel, "Evaluating a Framework for Different Networking Paradigms", in *Proc. of the 38th IEEE Conference on Local Computer Networks (LCN 2013)*, Sydney, Australia, Oct. 2013.

[3] D. Martin *et al.*, "Netlet-based Node Architecture Project Homepage.", [Online]. Available: http://nena.intend-net.org/

[4] L. Popa, A. Ghodsi, and I. Stoica, "HTTP as the Narrow Waist of the Future Internet", in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets 2010)*, Monterey, CA, USA, 2010.

[5] J. Ubillos, M. Xu, Z. Ming, and C. Vogt, "Name-Based Sockets Architecture", I-D (draft-ubillos-name-based-sockets-03), Mar. 2010.

[6] M. Welzl, S. Jörer, and S. Gjessing, "Towards a Protocol-Independent Internet Transport API", in *Proc. of the 4th International Workshop on the Network of the Future (FutureNet IV)*, Kyoto, Japan, Jun. 2011.

[7] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldine, and A. Nagurney, "Choice as a Principle in Network Architecture", in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Helsinki, Finland, Aug. 2012, pp. 105–106.

[8] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, "XIA: Efficient Support for Evolvable Internetworking", in *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, USA, Apr. 2012.

[9] T. Koponen, S. Shenker, H. Balakrishnan, N. Feamster, I. Ganichev, A. Ghodsi, P. B. Godfrey, N. McKeown, G. Parulkar, B. Raghavan, J. Rexford, S. Arianfar, and D. Kuptsov, "Architecting for innovation", *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 3, pp. 24–36, Jul. 2011.

[10] H. Wippel and O. Hanka, "End User Node Access to Application-Tailored Future Networks", in *Proc. of the 21st International Conference on Computer Communication Networks (ICCCN 2012)*, Munich, Germany, Aug. 2012.

[11] H. Wippel, T. Gamer, C. Faller, and M. Zitterbart, "Hierarchical Node Management in the Future Internet", in *Proc. of 4th Intl. Workshop on the Network of the Future (FutureNet IV)*, Kyoto, Japan, Jun. 2011.

[12] D. Martin, H. Wippel, and H. Backhaus, "A Future-Proof Application-to-Network Interface", in *Proc. of the International Conference on the Network of the Future (NoF 2011)*, Paris, France, Nov. 2011.

[13] B. Ahlgren, M. Brunner, L. Eggert, R. Hancock, and S. Schmid, "Invariants: A new design methodology for network architectures", in *Proc. of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA'04)*, Portland, OR, USA, 2004, pp. 65–70.

[14] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content", in *Proc. of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2009)*, Rome, Italy, Dec. 2009.

[15] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard, "VoCCN: Voice over Content-Centric Networks", in *Proc. of the 2009 Workshop on Re-architecting the Internet (ReArch'09)*, Rome, Italy, Dec. 2009.

[16] K. Scott and S. Burleigh, "Bundle Protocol Specification", RFC 5050 (Experimental), Internet Engineering Task Force, Nov. 2007.

[17] IBM and Eurotech, "MQTT V3.1 Protocol Specification", Aug. 2010. [Online]. Available: http://mqtt.org