Karlsruhe Institute of Technology (KIT)
Institute of Telematics

TELEMATICS TECHNICAL REPORTS

API Usage and Message Passing in NENA

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

January 3, 2013

TM-2013-1

ISSN 1613-849X

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

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

API Usage and Message Passing in NENA

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

Abstract—**The design and specification of new network protocols that aim at solving issues of currently deployed ones is a long, tedious process. It requires Internet-wide consensus and often leads to compromises. Although stable and interoperable implementations of protocols are part of the protocol specification process, new protocols suffer from another problem: acceptance and use by application developers. In order to solve this problem, we proposed NENA, a framework that aims at a better decoupling of applications and networks. It allows to run multiple network stacks in parallel, completely transparent to applications. In addition, it provides an implementation environment suitable for component-based protocols which offers ideal conditions for protocol composition approaches. In this paper, we describe aspects of our implementation that provide the necessary abstractions for NENA's goals: a protocol and network agnostic API relieving the application programmer from networking specific decisions, and a message passing system for component-based protocols with flow control capabilities that interact with flow control mechanisms of network protocols.**

I. INTRODUCTION

The evolution of the Internet from a small network of static computers used to transfer electronic mail and small files to a worldwide network with millions of nodes and thousands of different applications poses a challenge for the current Internet architecture. The current Internet architecture was not designed for mobility, security, group and real-time communication, or special use cases like sensor-actuator networks and machine-to-machine communication, which are becoming more and more relevant today. Additionally, the introduction of new features in the network is difficult which is shown by the slow adoption of new protocols like SCTP and IPv6 or the lack of global multicast services or Quality-of-Service guarantees.

These problems gave birth to many Future Internet research projects like the G-Lab project [1]. Within the G-Lab project, we worked on a Future Internet scenario in which nodes do not connect to a single, general-purpose network. Instead, nodes connect to different networks, each tailored to a specific application or use case. These application-tailored networks are based on individual network architectures and use network protocols that are optimized for the respective application or use case. 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). In this Future Internet scenario, a node connects to a network dynamically at runtime whenever the node accesses content or services that are offered within the network. As part of this process, an end-system acquires all necessary protocols and sets up a link to the network in order to eventually access the content or service.

Network virtualization and protocol composition are enablers for this scenario. Network virtualization [2] allows us to instantiate logical networks on top of existing network infrastructure. The operational benefit of network virtualization is that networks can be instantiated and modified on demand and relatively fast (compared to the installation/modification of physical network infrastructure). Additionally network virtualization provides technical benefits. Virtual networks can be operated in parallel and are isolated from each-other. This means network virtualization can improve resource utilization (due to statistical multiplexing) and errors in one virtual network do not impact other virtual networks. Additionally, virtualization provides the flexibility to adapt and migrate virtual resources if the underlying infrastructure changes, e. g., due to maintenance or failures.

Protocol composition (where some recent approaches are, e. g., [3], [4]) allows for a modular design of protocols by utilizing building blocks (BBs). This allows to re-use existing BBs as well as eases the testing and exchange of protocol functionalities if a functionality provided by a BB does not fulfill its expectations. Ideally, this concept enables a protocol design in which the protocol designer describes the information and control flow between (existing) protocol mechanisms rather than (re-)implementing everything from scratch.

As an experimental environment for such abstractions, we developed the runtime framework NENA (Netlet-based Node Architecture, [5]). NENA is executed on each network node and allows to operate different networks and their respective protocol architectures in parallel. Applications use NENA as a replacement for the current network stack. Applications communicate through NENA and thus initiate communication with remote services or content by using its API. The API increases the communication abstractions by hiding network details from the applications: application programmers no longer have to deal with name-to-address resolution, protocol selection, or multi-protocol support.

In this implementation-oriented paper, we present our experiences developing the run-time framework NENA itself. We show the challenges we faced in this process and our solutions. Furthermore, we describe open issues and indicate possible solutions we are tackling in current and future work.

The remainder of this paper is structured as follows: Section II gives a short overview of the Future Internet scenario that we are considering in order to point out the relevance of the multi-network approach. Section III outlines

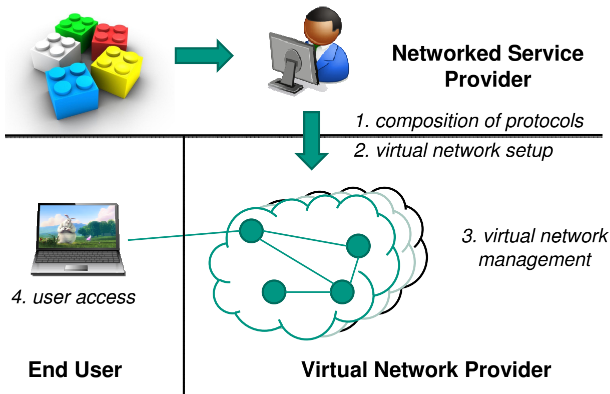

Fig. 1. Future Internet scenario overview.

the architecture of the NENA framework. Then, we put two aspects of the NENA framework into the focus of this paper: The implementation of the application interface that replaces the socket API (Section IV) and the implementation of a message passing system with networking-specific flow control (Section V). Section VI introduces a simple architecture serving as an example for a network protocol suite implemented within NENA. Finally, we recap design decisions and issues in Section VII.

II. Scenario

In the global picture introduced in the previous section, the following main actors are involved (Figure 1): The Service Provider, the Virtual Network Provider, and the End-User. The Service Provider (SP) creates an online service and designs a complete network solely for this service. Based on his requirements, he selects suitable networking paradigms, corresponding protocols, resources, and topologies. The Virtual Network Provider (VNP) creates virtual networks based on the resource and topology requirements given by the SP. He constantly monitors the resource usage of the virtual networks and may optimize the usage of his infrastructure based on the monitored data of all networks he runs. The End-User (EU) is the service customer: accessing the new service must be as easy as possible for him, preferably using user interfaces and applications he is accustomed to.

When an end user wants to access the service that is offered in a new application-tailored network, the end user's node retrieves meta-information about the new network from a global mapping service based on an URI. Then, the end user node establishes a virtual link to the virtual network. After this, the end user node queries the required components (protocols) from the mapping service. Using the retrieved information, the node then downloads, validates, and instantiates the protocols, and the new network is ready to be used by any appropriate user application. A more detailed description of this process can be found in [6] and [7].

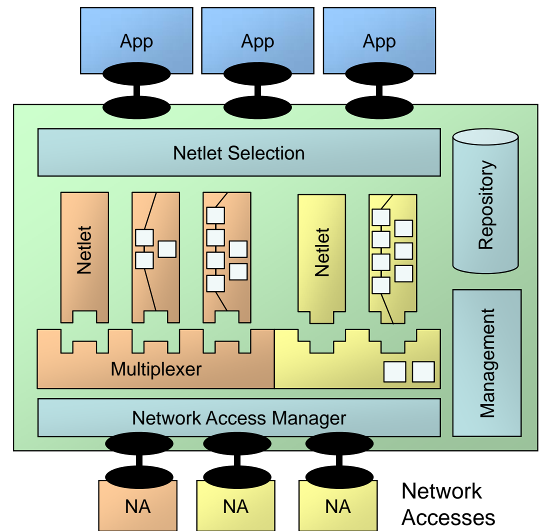

Fig. 2. Netlet-based Node Architecture (NENA)

III. Framework (NENA)

In this section, we present an overview of the NENA framework and its relevant components. For more details about the concepts, please refer to [8].

NENA (Figure 2) is a runtime framework that allows nodes to simultaneously connect to multiple networks at the same time. These networks can be based on different network architectures using specialized network protocols. Application access is provided by an API that abstracts from networking details: Instead of providing network addresses and protocols, applications specify globally unique names as URIs to initiate communication with the content, service, or host associated with the name. NENA then performs network and protocol selection transparently to the application. Additionally, applications can influence this selection by specifying application requirements via the API. These requirements can, for instance, contain Quality-of-Service parameters, specify the requested data format (i.e., content-type), or simply specify traditional requirements such as reliable transport. More information about the API can be found in Section IV.

Protocols or complete protocol stacks in NENA are encapsulated in so-called *Netlets*. The Netlet Selection component performs the selection of networks and protocols. The requested name is used to select a suitable network, while the application requirements are used to select a suitable protocol (Netlet). This is a two-step process, where first each architecture is inquired to deliver a set of Netlet candidates that are able to fulfill the request. This can be seen as an filtering process that takes the URI and the mandatory requirements of the request into account. Second, the "best" candidate is chosen based on optional requirements and network properties (such as bandwidth or delay) if applicable.

Below the Netlet Selection component, the components of each currently active network are located. Multiple networks with individual architectures can be in use on a node at

the same time. Each network architecture instance consists of a set of Netlets, a multiplexer, and a set of network accesses. An architecture multiplexer constitutes the base-layer of a network stack, performs Netlet multiplexing, and – depending on the architecture – implements addressing and forwarding mechanisms. An architecture 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 (NIC). Thus, multiplexing of multiple virtual networks over the same physical network has to be realized outside of NENA (e. g. via VLANs). The Network Access Manager controls the creation and removal of network accesses and notifies the respective multiplexers. Although the Network Access Manager is depicted as an intermediate layer, multiplexers and network accesses communicate directly once they are associated with each-other.

For management and maintenance purposes, NENA also has a network independent repository and management components. The repository component loads and instantiates Netlets and multiplexers from external libraries. The management component is the central entry point for management requests, which means that it propagates requests to the respective architectures and collects monitoring information from different levels (node-wide, architecture-specific, or protocol-specific) [9].

IV. API

A crucial part of the scenario described at the beginning of the paper is to allow independent evolution of both, applications and networks. The design goal of the NENA API is abstracting from network details. This means, that the required networking knowledge like address formats and protocols is removed from the applications and pushed down below the API. Furthermore, the API should be generic enough to cope with different communication paradigms like host-based or content-centric communication. The API should be flexible with respect to the introduction of new networks and protocols. Both of which should not require to change existing applications. In order to achieve these goals, we use globally unique names and moved name resolution and protocol selection below the API.

A. Primitives

Based on the proposals in [10] and [11], we implemented an API fulfilling the before-mentioned goals. Its usage pattern is similar to today's socket API: a communication end-point is created with a primitive that returns a handle on which read/write operations may be performed. This basic pattern has proven to be very portable. A C-library can be created easily and eventually integrated in implementations for other programming languages. In addition, existing applications can be ported from the socket API to the NENA API without changing much in the I/O logic. Any high-level abstractions such as callback-based interfaces for event-driven applications can be realized with the destination languages' means (as it is done today with the socket API).

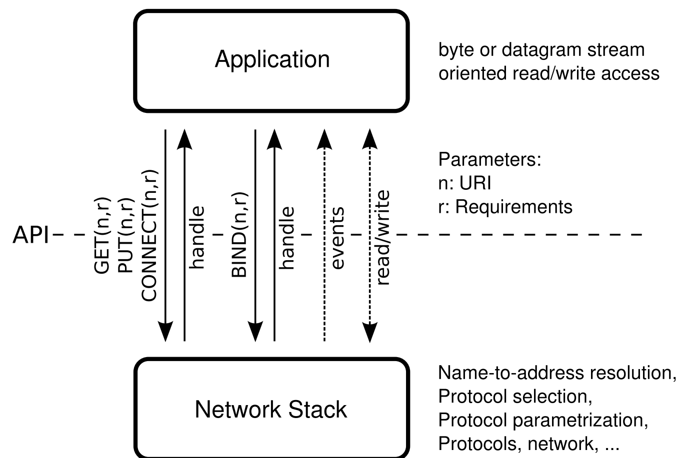

Fig. 3. API Overview.

Instead of providing only one primitive to create a communication end-point, we provide three (Figure 3): CONNECT, GET, and PUT. While CONNECT matches today's socket() semantics closest, GET and PUT are introduced in order to support the many contemporary applications, that use a RESTful interface (e.g., HTTP) as their basic communication abstraction [12]. By providing those methods at the API level, network services are no longer bound to use HTTP as their basic communication abstraction, and eventually new communication paradigms such as Content-Centric Networking [13] can be easily introduced.

The application providing service or content as a server uses BIND in order to announce its availability to serve requests. Instead of well-known port numbers to identify services, the application specifies an URI. Here, wild-cards or longest-prefix matching allows an application to register itself for a base-URI. Requests with URIs containing this base-URI are also 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). Currently, the implementation only supports byte-stream-based end-points, but extensions to allow datagram-based end-points (with arbitrary application data unit lengths) are planned.

B. Implementation

The primitives and supporting procedures were implemented in C++ as a library which also provides a C-interface. The library deals with the NENA IPC protocol to transport data and meta-information between NENA and applications. The requirements/properties are currently 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
}.
```

While this introduces an overhead due to text parsing, it provides extensibility – a main concern for experimentation environments. This overhead, however, is only introduced during setup of an end-point. On-going communication over an existing communication association is not affected.

The C-API allowed us to create an object-oriented API in Python. The Python API loads the C-library and builds an object-oriented wrapper around it. Handles are realized as objects. Thus each of the handle creating methods GET, PUT, CONNECT, and ACCEPT returns a handle object. The handle object then offers the methods operating on the handle. These methods are read, write, wait, and accept.

The Python API was used to create an HTTP-based API wrapper (WebAPI). We used a multi-threaded Python web server implementation which is bound to localhost. In this implementation the HTTP-Get request is redirected to the WebAPI by using a Browser-Plugin. This Plugin sends HTTP-Requests to the pre-configured local WebAPI instance based on the URL scheme name (for example nenaweb://). The application requirements are embedded as parameters in the requested URL (for example nenaweb://kit.videostore/index.html?content-type=text/html). The WebAPI maps HTTP-Get requests to NENA-GET requests. In this process, the requested URI and the requirements are extracted from the URL. The WebAPI then forwards the data retrieved from the NENA API handle to the application (a web browser) as HTTP-Reply. Opposed to implementing a protocol handler directly in an open-source web-browser, this proved to be more portable and easier to maintain.

The NENA API is also used as a Management API to ease management tasks of the NENA node. Management information is accessed in a hierarchical management name space (e. g., nena://<host>/<componentType>/<componentID>). In its current implementation it is used to retrieved information about active Netlets, multiplexers, application connections, and network accesses on a NENA node. The GET request for "nena://localhost/netadapt/eth0", for instance, returns

```
{
  "maximumBps": <maximum bandwidth>,
  "rxRate": <current RX rate>,
  "txRate": <current TX rate>,
  "carrier": <carrier detected 1/0>
},
```

i. e. some statistics on the network access provided by eth0.

C. *Usage Recommendations*

Once we built this API, it was much easier to develop demo and test applications, since they didn't need to implement the NENA IPC protocol any longer. With the Python API implementation, it was even easier to create test applications or demo applications.

One such example is a minimalistic file-server implementation in Python. The file-server only needs to bind itself to a base URI, e. g. file://kit.videostore/files, enter its main loop and wait for new client requests. Upon an incoming request, a new handle is created with ACCEPT. The request-URI is read from the handle's meta-information, e. g. file://kit.videostore/files/videos/movie.webm. From this URI, the file-server strips the base-URI and reads the requested file from a pre-configured directory. The benefit here is, that the file-server does not need to implement an application-layer protocol, since the GET/PUT semantics are already part of the API.

Using the name-based approach, content-centric paradigms can be easily supported. For content retrieval, GET is used. To publish content, there are two possibilities: With PUT, the content can be put into the nearest cache (which could simply be the cache on the local machine) and the application is not further involved regarding future requests. If the content should not be put proactively into the network, the application can choose to BIND on the content's name. After the initial retrieval, the application is only involved again if the content is preempted from any network cache. To improve interactive communication via a Content-Centric Network (CCN), the applications may provide requirements indicating which caching policies should be used. However, we would recommend to use another network architecture that better caters to the needs of interactive applications.

An interface paradigm often used to access CCNs is Publish/Subscribe. In a traditional message-broker architecture, however, we recommend a different usage of our API compared to CCN: To publish a new message to a topic, PUT is used. The topic's name (an URI) may contain a specific message broker if the architecture requires it. To retrieve the last message published to a topic, GET should be used. In order to subscribe to a topic and to retrieve continuous notifications, a CONNECT to the topic's name should be used. Note, that in this case, BIND is not used at all by client applications. If the message broker, however, is realized at the application layer, it can announce its presence to the network using BIND.

Since BIND is network architecture independent, the actions necessary to actually bind an application connection to an URI may differ per network. In a traditional TCP/IP network, it could simply result in opening a well-known port: The URI http://localhost, for instance, would open TCP port 80. Within this namespace, however, applications would need to implement HTTP. We thus recommend introducing another namespace such as www:// which denotes that the content should be part of the World-Wide Web. An appropriate Netlet

implementing HTTP will then translate the API primitives to HTTP primitives and vice-versa. More sophisticated announcement services may be integrated as well, such as via DNS service records [14], Bonjour (i. e., mDNS [15] with service records), or DHTs. The methods best suited for the respective network architecture can be chosen by the architecture itself.

V. Message Processing

While the API described in the previous section increases the abstraction level for the application programmer, the message processing system described in this section aims at increasing abstractions for the protocol implementer. Since networking at its base level is about message, event, and timer processing, we used a message passing system in NENA. Hence, each protocol mechanism (or protocol building block, BB) is realized as message processor and a protocol consists of a set of message processors which interact using messages and events.

A. Message Passing

The message passing system in NENA uses *message schedulers* (MS) and *message processors* (MP). A MS hereby manages a set of MPs. MPs are the base class for all components in NENA, in particular for Netlets and protocol building blocks. The MS maintains message queues for each of its MPs. For each message, the MS calls a processing function of the destination MP. When a MP sends a message to another MP, the message is put into the message queue of the destination MP, and execution of the current MP continues until its processing function returns. Then, the MS calls the processing function of the next MP. Currently, round-robin scheduling is used.

At this level, no locking is required, since a MS and all of its MPs run within the same thread. Multi-threading is supported by adding additional message schedulers, each running in its own thread. When messages are exchanged between MPs of different MSs, a synchronization is required which essentially means that locks for message queues are necessary.

The number of MSs and the distribution of MPs to MSs is currently configured statically. This architecture, however, allows the distribution of MPs across different threads depending on their run-time behavior and their interaction with other MPs. This, however, is subject to further investigation.

In addition to directly addressed messages, timer messages and event messages are distinguished. Timer messages are self-messages which are delayed by a given time. Event messages are messages exchanged via the observer pattern: A MP defines a set of events other MPs can register to. When an event occurs, the emitting MP notifies all of its listeners.

For this message passing based approach, two additional implementation concepts have proven indispensable: shared pointers and scatter-gather buffers. In a message passing system, messages are created, duplicated, and consumed by different MPs. Destroying messages manually once they are consumed bears the potential risk of destroying messages to

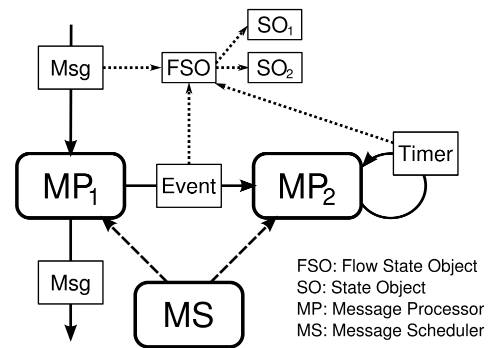

Fig. 4. Message passing components and their relations. Messages, events, and timers exchanged between message processors (MPs) hold references to flow state objects (FSOs). A message scheduler (MS) manages a set of message processors and calls their processing functions.

which other MPs still hold references. Similarly, it is easy to "forget" about destroying a message which is no longer needed, and thus introducing memory leaks. To avoid these issues, we use shared pointers which are an implementation of reference counted pointers in C++/boost. This introduced only a small overhead while providing convenience for the programmer and increasing stability.

Scatter-gather buffers are an important concept especially when using a lot of fine-granular protocol building blocks that add micro headers. Initially, we experimented with preallocated buffer spaces, writing header fields and moving cursors around. Despite helper-methods it has proven to be too tedious to use and to debug. With the introduction of the reboost message class [16] developed for ariba [17], headers are now serialized into their own micro buffers and prepended to the message buffer list. The message buffers can still be iterated sequentially and byte-wise if needed (for instance, to compute a CRC). They are linearized only when sent to the network. Another feature of the reboost message class is a copy-on-write implementation. If messages are duplicated, only the message class is copied while the payload buffers are shared by all instances of the message until they are modified.

B. Flow States

For each application connection, a *flow state object* (FSO) is created. Its basic purpose is similar to a TCP Control Block (TCB) but its scope is extended. It holds basic information about the flow such as the requested remote URI and the request method (i. e. CONNECT, GET, PUT, BIND). A FSO is uniquely identified by a flow ID. Together with the host's name, a flow ID identifies an end-point for a communication association. Unlike port numbers, flow IDs do not represent well-known services and must be considered random. Service identification is solely based on the provided URI (see Section IV-A).

The FSO uses a slightly modified observer pattern: The FSO provides events to which message processors can register to. Notification to registered MPs, however, is not triggered by the FSO itself, but by other MPs. Examples of such events are changes of the operational state of the FSO: If

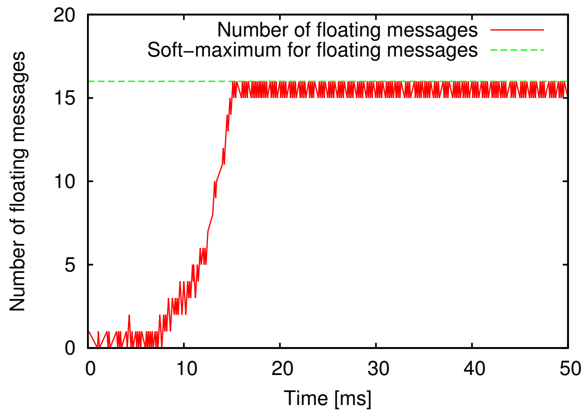

Fig. 5. Number of floating messages with an unreliable datagram transport. Once the soft-maximum is reached, the counter is decreased and increased in an alternating way.

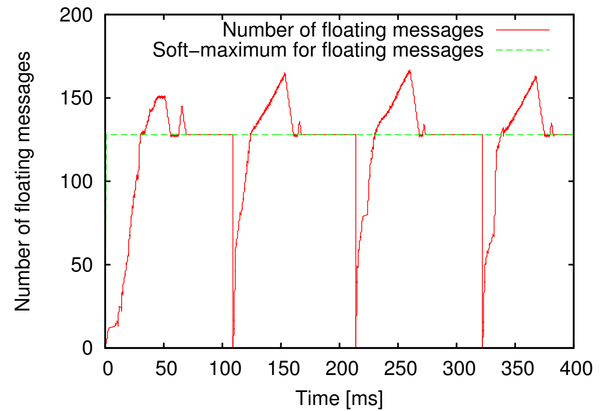

Fig. 6. Number of floating messages with a Go-Back-N ARQ (N=128, delayed cumulative acknowledgments after 100 ms). Due to message buffering in the retransmission buffer, the soft-maximum may be exceeded temporarily.

the communication ended regularly (or got disrupted), the operational state is changed from *valid* to *ended* (resp. *stale*). The MP changing the operational state then notifies all MPs registered to this event.

Besides basic flow information, the FSO can hold additional *state objects* (SO). SOs are created by MPs and hold per-flow state information such as sequence numbers. This allows to create MP objects which by themselves are stateless. Therefore, it is possible to update MP instances on-the-fly without disrupting ongoing communication (as long as the new MP version does not introduce a different SO). Figure 4 summarizes the relations between message passing components.

C. Floating Messages

Messages traveling through NENA between applications and network encounter several buffers and queues. Each NENA component is a message processor (MP) and has a message queue for messages directed to it. Those message queues are not limited in size since this would require proper handling of exceeded queue limits without dropping messages. The only way to enforce this, would be to halt execution of all MPs upwards all message chains leading to the congested MP. This may lead to deadlocks since message chains are not required to be loop free (note, that message queues not only include data packets but also event and timer messages). To resume message chains, a scheduling strategy would be required to retain fair behavior.

To keep things simple, NENA uses another approach which allows unlimited queues per MP. Messages between application and network are called *floating* messages, i. e. messages within NENA which were not yet sent to the NIC. Messages not yet acknowledged from the remote NENA instance are called *flying* messages. Flow and congestion control mechanisms of network protocols generally limit the flight-size of messages per flow. Similarly, the float-size can be limited per application flow on a single NENA instance (independently of the number of MPs and queues the flow passes through). This is achieved by incrementing/decrementing counter variables of the flow state object each time a message is generated, duplicated, or consumed. Once a certain maximum is reached, no more data is taken from the application's IPC socket which eventually blocks the application. Thus, rather than having a traditional buffer for application data, it can be considered as a virtual buffer distributed across several MPs and their respective queues and buffers (such as retransmission buffers).

This mechanism interacts with window-based flow and congestion control mechanisms of network protocols: The number of floating messages can be limited to their current window sizes (which also determine the retransmission buffer sizes). However, the maximum of floating messages can only be treated as a soft-maximum, which means that it may need to be exceeded temporarily (to twice the soft-maximum in the worst case). This is the case when a retransmission buffer is full (thus, the maximum of floating packets is reached) and a retransmission is triggered: For each retransmitted packet, the counter for floating messages will grow by 1 beyond the currently allowed maximum until it gets consumed by the network device. However, no new messages from the application are generated. Once the number of floating packets falls below its current soft-maximum, a flow state event is triggered which allows NENA to read again from the application's IPC socket.

Figure 5 shows the value of the floating packet counter with a simple unreliable datagram transport. In this scenario, NENA was slowed down to become a processing bottle-neck in order to visualize when NENA blocks application I/O. Since no message is duplicated within NENA, the floating message maximum (which was set to 16) is never exceeded. The ramp-up time at the beginning of the communication is due to system I/O buffers filling up (application IPC and network device).

Figure 6 shows the value of the floating packet counter when using a Go-Back-N ARQ building block with a window-size of N=128 packets and cumulative acknowledgments which are delayed by 100 ms. In this case, the allowed soft-maximum of floating packets is set to Go-Back-N's window-size which is 128 packets. The number of floating packets, however, still increases beyond the soft-maximum since additional refer-

ences of packets are stored in Go-Back-N's retransmission buffer in addition to sending them further down the stack. The two peeks in each round are due to thread scheduling behavior since the processing of the Go-Back-N building block (and thus the addition of a reference to the retransmission buffer) and network I/O happen in different threads. Once all unsent packets are sent to the network the number of floating packets matches their soft-maximum until an acknowledgment is received. Then, all acknowledged packets are purged from the retransmission buffer. Since the number of floating packets is now below its soft-maximum, the application is unblocked and new data can be sent. Naturally, this idle time before an acknowledgement is received should be minimized with a proper configuration of Go-Back-N or by using mechanisms with adaptable window sizes. Transport protocols with variable window-sizes can continuously adapt the soft-maximum of allowed floating packets according to their window-sizes. If multiple window-sizes influence the soft-maximum (e. g. from flow control and congestion control mechanisms), the smallest value must be used.

If there are no limiting factors such as window-sizes from network protocols, the soft-maximum should be chosen as small as possible to reduce end-to-end delay for applications. The more packets are floating within NENA, the more delay is added before they are actually sent to the network. On the other hand, the value must not be chosen too small to avoid unnecessary processing idle times. If set, for instance, to 1, it would behave like a Stop & Wait flow control mechanism within NENA with its accompanying inefficiency.

A drawback of the counter approach is, however, that each MP is required to increment/decrement the counter variables if it creates additional messages or consumes messages. For complex mechanisms, this might yield to implementation bugs where the counter does not match the number of messages and thus stalls the communication (if it is increased too often) or allows more and more messages within NENA (if it is decreased too often). To deal with this issue, we are currently investigating to what extent the reference count of the shared pointer concept can be exploited to increase/decrease those counters automatically. The use of message class pools (which hold references to messages for later reuse) currently disallows this approach.

VI. A SIMPLE NETWORK ARCHITECTURE

In this section, we will give an overview of our "Simple Architecture" which we used to experiment with our framework. While it is not sophisticated, it should serve as an example of the necessary decisions that need to be made when designing a new architecture within the NENA framework.

A. URI Scheme

Each architecture and Netlet has to define the URI schemes it supports. These can either be proprietary schemes coming with the architecture or global schemes registered at a global authority such as IANA. Format and semantics for those schemes need to be standardized in this case.

For our Simple Architecture, we defined two proprietary schemes, node:// and app://. Node-URIs identify hosts, app-URIs identify application services. Since name-to-address resolution in Simple Architecture (see next section) requires node-URIs, app-URIs are mapped to node-URIs. To make this mapping canonical, the app-URI has the following format: app://<hostname>/<servicename>/<serviceparameters>. Such URIs are then mapped to node://<hostname>.

B. Addressing, Neighbor Discovery, Forwarding

One requirement for the Simple Architecture was that it runs over UDP tunnels, since this is the easiest way to set up communication in demos and on PlanetLab-based testbeds. Hence, we chose UDP end-point addresses as interface addresses for network accesses, i. e. <IP address>:<port>. However, there is no differentiation made between IP address and port number, thus the complete tuple serves as a locator. Additional network accesses can then be created easily via additional UDP end-points, and multiple NENA instances can run on the same node. The latter is a feature which can be used to set up scenarios similar to [18], where a whole network can be emulated on a single node with realistic operating system behavior.

The multiplexer builds a base layer for this architecture. On an end-system, it performs name-to-address translation, which means that late binding of names to addresses is realized here: transport protocols are not aware of any network addresses and only work with URIs supplied by the application. In fact, in most cases, protocol mechanisms do not even need the URI since flow multiplexing is already done by the framework by providing a flow state object with each message. After name-to-address translation, the multiplexer needs to determine an appropriate network access over which a message needs to be sent. For this, it possesses a forwarding information base (FIB) containing next-hop addresses for known addresses. On intermediate-nodes, the multiplexer uses this FIB for forwarding messages.

The FIB is populated by a "Routing" Netlet which performs simple neighbor discovery and neighbor information exchange. There is no routing algorithm or path metric implemented, nor are routes tested if they are still valid. Neighbor discovery is done via broadcast messages sent over each network access. Broadcasts can either be done as UDP broadcasts (which has the disadvantage that they may not be allowed in certain networks) or via a configurable list of UDP end-points where the "broadcast" message should be sent.

C. Service, Flow, and Netlet Identification

Service and Netlet IDs need to be sent over the network in order to allow the remote multiplexer to determine the correct Netlet and Service. Since service URIs and Netlet IDs (also represented as URIs) are relatively long strings, only a hash is sent to the remote node. Upon receiving a message, the hashes are compared to the hashed strings of known services and Netlets and appropriately forwarded to the respective components.

In each message, the local flow ID and the remote flow ID (as far as it is known) is sent. Together with the nodes' names, the quadruple (<node$_1$>, <flow$_1$>, <node$_2$>, <flow$_2$>) uniquely identifies an application flow. Note, that at this point the nodes' names are used and not their addresses. This allows flows to survive in case of changing network access addresses.

Once the flow IDs of both nodes are known, flow state objects on the respective nodes can be determined without comparison of service or Netlet hashes.

D. Transport Netlets

We have built two simple transport Netlets: a reliable and an unreliable one. Both are composed of protocol building blocks that are architecture independent. While the unreliable Netlet only consists of a segmentation building block, the reliable transport adds a Go-Back-N ARQ building block and a building block to signal REST commands. Thus, the reliable transport Netlet is the only one supporting the full feature set of the API, while the unreliable one only supports CONNECT and PUT in a basic way without sending service parameters to the remote node. GET requests and service parameters require a reliable transmission and, thus, the REST building block was not integrated into the unreliable transport Netlet. During the Netlet selection process in NENA, this and any additional application requirements (e. g. requested reliability) are checked. If both Netlets are able to serve a given request, precedence is given to the unreliable Netlet.

VII. Conclusion

The implementation concepts described in this paper partially use software-engineering patterns which are only offered by high-level programming languages such as C++ (with STL and boost libraries) and Java. In OS kernel environments, such languages are not always available, which is why our framework is running completely in user space. While this may not be suitable for high-performance network devices such as switches and routers (which anyway rely on hardware implementations), end-systems and middle-boxes may benefit from user space approaches. In fact, some kernel drivers find themselves today in user space in all major operating systems. User space approaches with high-level language support yield faster development cycles, more frequent updates, and a huge diversity – provided that the right abstractions were made to increase acceptance and ease-of-use. With NENA, we gained implementation experience with such abstractions. Over the years, the framework grew and was used for experimentation, demonstrations, student theses, and student lab assignments. While interfaces now allow much faster additions of applications and protocol building blocks than at the beginning, there still are a few inconveniences left – which mostly is because of the prototype character, though.

The huge palette of HTTP/Web-based applications today suggests that the current socket interface is not the best abstraction for application programmers. With the API implementation presented in this paper, we showed that the key properties of what HTTP provides (i. e. name-based GET/PUT semantics) can be moved down below the API without limiting the application to the HTTP/TCP/IP trio. This still allows the continuation of innovations at the application-level while, in addition, enables innovation in the network. With a message and event-based decoupling of protocol mechanisms, the latter becomes even easier to realize since mechanisms evolved over decades can be used for different protocols via protocol composition approaches.

References

[1] "G-Lab Project Homepage.", [Online]. Available: http://www.german-lab.de/

[2] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization", *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[3] R. Dutta, G. N. Rouskas, I. Baldine, A. Bragg, and D. Stevenson, "The SILO Architecture for Services Integration, controL, and Optimization for the Future Internet", in *Proc. of the IEEE International Conference on Communications (ICC 2007)*, G. N. Rouskas, Ed., 2007, pp. 1899–1904.

[4] J. Touch and V. Pingali, "The RNA Metaprotocol", in *Proc. of 17th International Conference on Computer Communications and Networks (ICCCN 2008)*, St. Thomas, Virgin Islands, USA, Aug. 2008.

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

[6] O. Hanka and H. Wippel, "Secure deployment of application-tailored protocols in future networks", in *Proc. of the International Conference on the Network of the Future (NoF 2011)*. Paris, France: IFIP/IEEE, Nov. 2011.

[7] H. Wippel and O. Hanka, "End user node access to application-tailored future networks", in *Proceedings of the 21st International Conference on Computer Communication Networks (ICCCN 2012)*, Munich, Germany, Aug. 2012.

[8] 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.

[9] H. Wippel, T. Gamer, C. Faller, and M. Zitterbart, "Hierarchical node management in the Future Internet", in *Proc. of 4th International Workshop on the Network of the Future (FutureNet IV)*. Kyoto, Japan: IEEE, Jun. 2011.

[10] F. Liers *et al.*, "GAPI: A G-Lab Application-to-Network Interface", in *Proceedings of the 11th Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop "Visions of Future Generation Networks" (EuroView2011)*, Würzburg, Germany, Aug. 2011.

[11] 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: IFIP/IEEE, Nov. 2011.

[12] L. Popa, A. Ghodsi, and I. Stoica, "HTTP as the narrow waist of the future Internet", in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets 2010. New York, NY, USA: ACM, 2010.

[13] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content", in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. New York, NY, USA: ACM, 2009.

[14] A. Gulbrandsen, P. Vixie, and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, Feb. 2000.

[15] S. Cheshire and M. Krochmal, "Multicast DNS", draft-cheshire-dnsext-multicastdns (Internet-Draft), Dec. 2011.

[16] S. Mies, "Reboost buffer classes.", [Online]. Available: https://github.com/sebastian-mies/reboost

[17] C. Hübsch, C. Mayer, S. Mies, R. Bless, O. Waldhorst, and M. Zitterbart, "Reconnecting the Internet with ariba – Self-Organizing Provisioning of End-to-End Connectivity in Heterogeneous Networks.", *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 131–132, Jan. 2010.

[18] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks", in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2010.