A System Architecture for Evolving Protocol Stacks

(Invited Paper)

Ariane Keller, Theus Hossmann, Martin May
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
Email: firstname.lastname@tik.ee.ethz.ch

Ghazi Bouabene, Christophe Jelger, Christian Tschudin
Computer Science Department
University of Basel, Switzerland
Email: firstname.lastname@unibas.ch

***Abstract*—A majority of network architectures aim at solving specific shortcomings of the original Internet architecture. While providing solutions for the particular problems, they often lack in flexibility and do not provide general concepts for future networking requirements.**

In contrast, we introduce a network architecture that aims to be versatile enough to serve as a foundation for the future Internet. The main pillars of our architecture are communication pivots called information dispatch points (IDPs) which embed the concept of modularity at all levels of the architecture. IDPs completely decouple functional entities by means of indirection thus enabling evolving protocol stacks. Our architecture also provides a consistent Application Programming Interface (API) to access node-local or network-wide functionality.

In addition to the description of this architecture, we report about a working prototype of the architecture and we give examples of its application.

I. Introduction

The success of the existing Internet architecture is a testimony of the wise design decisions [1] of the early days of the Internet. Indeed, the closely specified protocol suite and the simple basic mechanisms paved the way of this success.

However, for today's and future challenges, this architecture may not suffice: With the continuous growth of the number of networking devices and their increased diversity in functionality, the role and the properties of the networking elements become challenged.

Therefore, several research projects have tried to develop a network architecture that is armed for the future. Some of these projects focus on virtualization methods to separate router functionality or whole networks [2], [3]. Others discuss on a higher abstraction level how networks beyond IP can be built [4], [5], [6]. These papers discuss issues like naming and addressing or network pluralism in general.

It is foreseeable that future networks will be heterogeneous (in devices and functionality) and will require to be able to adapt dynamically in order to provide a protocol stack that is suitable for the actual network environment. A major difficulty in providing such a new architecture is the design of a generic mechanism that flexibly connects the different networking elements. Flexibility is typically achieved in computer science by adding a layer of indirection. In [7], the authors illustrate how indirection can be used in an elegant and clean distributed network architecture. Indirection as such is a well understood principle. For example, file descriptors in Unix systems are used as an abstraction mechanism for nearly everything that is able to read and write data. Sockets, device files, and the console are all treated the same way as physical files. This allows an application to send data to a file descriptor without having to worry whether the data is actually sent to a file or to the console. Such indirection even allows to redirect all data received by the console to a physical file in a *transparent* manner.

However, the gained flexibility comes at a price. The system gets more complex and the performance may be reduced due to some extra effort needed by the indirection. Therefore for each system a trade-off between flexibility on the one hand and performance on the other hand has to be considered.

Inspired by the flexibility provided by indirection, we have built a network architecture based on this principle: the ANA framework.[1] An initial discussion of ANA can be found in [8]. In ANA, we introduce indirection on *every* hierarchical level. Our architecture divides all network functionality into blocks, which are addressed by node-local identifiers. Instead of one function talking directly to another function, the first function sends its data to an identifier (called *Information Dispatch Point or IDP*) that is bound to the second function. The IDP simply has to forward the data to the second function.

Our architecture leads to a protocol stack that is ruled by IDPs. Each function, be it on the local or on a remote node, is accessed with the help of a local IDP. This layer of indirection allows changing the flow of data within a node at runtime and transparent to the sending block, thus the protocol stack can be optimized for the current network environment.

We have developed a generic API (Application Programming Interface) that enables communication between any pair of networking functionality. The main benefit of this API is that the same functions can be used for node-local as well as for remote communication setup.

Following the Internet tradition where networking software matures through implementation, we have implemented a prototype of our architecture. The knowledge gained during each implementation round has been used to improve the architecture and the API. As a result, our framework has matured on both, academic as well as implementation challenges.

A. Contributions

The network architecture we present herein is a framework for future networks. It differs in multiple aspects from existing

[1]The ANA project website: http://www.ana-project.org.

work as it is able to deal with the heterogeneity and evolution of networks as well as with the integration of new mechanisms and protocols at runtime.

Specifically, our architecture integrates the following new elements: (i) the *indirection* approach is applied for node local function decomposition; (ii) the system and network design allows for changing network stacks on the fly; (iii) our architecture does not imply the "one size fits all" approach. The system is conceived such that multiple stacks may coexist and allow for heterogeneous settings with network clouds with individual address and naming structures, packet formats, or routing schemes.[2] Finally, (iv) we provide a generic API to the networking mechanisms. This API provides flexibility and allows for evolution of the network without imposing any restriction to the internal network operation.

Due to (i) and (ii), one is able to integrate new functionality like for example monitoring or packet capturing in the node without service interruption. Also, the networking stack can be reconfigured transparently to better fit the networking condition. A typical example would be the use of additional functionality like encryption or authentication when a node enters an insecure network.

II. Core Networking Machinery

The fundamental concept around which the architecture is built, is the *information dispatch point (IDP)*. IDPs are inspired by the work on network pointers [9] and are also somehow similar to *file descriptors* and *sockets* in Unix systems. IDPs are typically bound to *functional blocks (FB)*. Functional blocks are information processing units that implement data transmission functionality (for example sending and receiving of IP packets) or some additional functionality as for example traffic monitoring. Generally, functional blocks can be used to implement network services like those described in [10]. IDPs also abstract *information channels (IC)* via which remote nodes and protocols are reached. However, unlike file descriptors and sockets, the binding of an IDP is dynamic and may change over time as the "network stack" is re-configured. From an implementation standpoint, an IDP is identified by a randomly generated label.

In order to keep track of all available functional blocks and IDPs, there is a central entity on each node. This entity has two tables: one that describes all functional blocks and a second that stores the mappings between IDPs and functional blocks. Those tables are used to forward messages between the individual FB.

A. Basic IDP operation

The objective of IDPs is actually two-fold: first, they provide a generic communication means between the various functional blocks running inside a node and, second, they provide the flexibility to re-organize the communication paths. Examples of such communication paths are illustrated in Fig. 1. In Fig. 1 (a), a functional block (FB1) sends data to an IDP 'a' which is bound to another functional block (FB2). In Fig. 1 (b), the IDP is rebound to the functional block FB3 and in 1 (c), the IDP is bound to the information channel IC1.

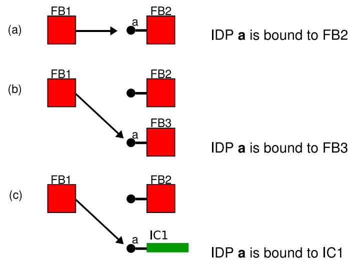


Fig. 1. IDP binding: IDP 'a' is subsequently bound to FB2, FB3 and IC1.

The important property of this re-binding operation is that it is not disruptive: in this example, any FB which was sending packets to IDP 'a' (e.g., FB1) continues to send to IDP 'a' without even being aware of the re-binding operation.

The IDP bindings are stored in a forwarding table within the node where each IDP is identified by a node-local label. This table, called the *information dispatch table (IDT)*, is illustrated by Fig. 2. As shown in the figure, the IDT stores the binding between IDP values and the entity (FB or IC) to which they are bound. When a packet is sent to some IDP, it is forwarded to the FB or IC to which the IDP is currently bound. It is then up to the entity receiving a packet to decide what to do next with the packet: consume the data, add a header and re-forward the packet to the next IDP, drop the packet, etc. For example, in Fig. 2, FB1 either forwards data to the IDP 'b' or directly to IDP 'c' (from which data is sent to some network interface for example). Note that a FB/IC may have multiple IDPs attached to it, as shown in the figure where IDPs 'y' and 'b' are both attached to FB2.

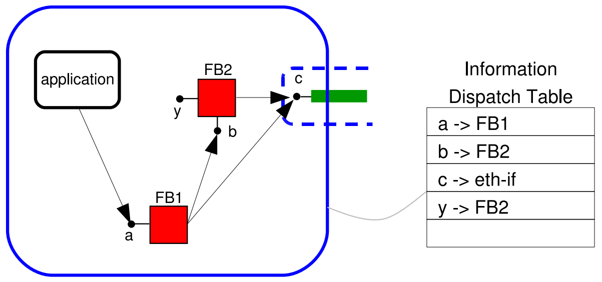
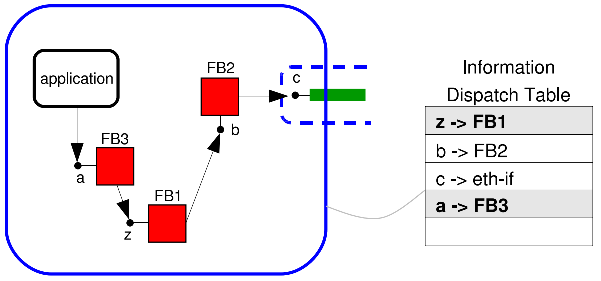


Fig. 2. Forwarding inside a node. The information dispatch table holds the bindings of the IDPs.

As previously stated, the binding of an IDP is not fixed and may be changed dynamically. For example, between Figures 2 and 3, the IDP 'a' was re-bound to functional block FB3. In order to perform a re-binding, only the entry of the particular IDP has to be changed in the information dispatch table. Note that the re-binding has been fully "transparent" to the application, which continues to send data to IDP 'a'.

For packet forwarding, IDPs permit to implement forwarding tables which are fully decoupled from addresses and

[2]For the interaction of these network clowds, an appropriate gateway functionality has to be provided.



z -> FB1
b -> FB2
c -> eth-if
a -> FB3

Fig. 3. IDP re-binding inside a node. Instead of sending data directly to FB1, data is send to FB3 and then forwarded to IDP 'z' bound to FB1

names: i.e., the next hop FBs (inside a node) and (remote) nodes are always identified by IDPs. This hence allows to add and use new networking technologies and protocols as long as they "export" their communication services as IDPs.

B. Advanced use of IDPs

The concept of IDPs is in itself very powerful and enables the development for multiple dynamic network element configurations. We further developed this concept by adding some sort of access control and scope to IDPs.

The major drivers for this add-on were: (i) functional blocks internally bind each IDP to a particular operation to be carried out and this means that IDP redirection must preserve IDP values; (ii) while performing an IDP redirection, one may not always wish to redirect all the packets sent to a particular IDP. For example, some networking stacks may wish to redirect only certain network traffic to an encryption FB while other packet flows are forwarded unencrypted. To support such functionality, we introduce two types of IDPs:

- *Public* IDPs: Public IDPs receive data from any functional block. They are identified by the subscript character $\star$ (such as e.g., $\mathtt{a}_*$).
- *Private* IDPs: Private IDPs only receive data from a particular FB or IC. This FB or IC is indicated as a subscript suffix (such as e.g., $\mathtt{a}_{FB1}$).

Fig. 4 illustrates how redirection is actually performed such that the functional block FB2 continues to receive data via the IDP 'a'. All data sent to the public IDP '$\mathtt{a}_*$' is now received by FB1 and then forwarded to FB2. At that point, the IDT has two entries $<\mathtt{a}_* \rightarrow \mathtt{FB1}>$ and $<\mathtt{a}_{FB1} \rightarrow \mathtt{FB2}>$ and dispatches packets to the appropriate destination according to the sender FB.

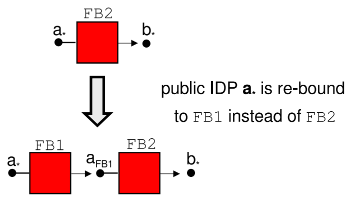


Fig. 4. Re-binding with public and private IDPs.

The use of public and private IDPs for performing selective redirection is illustrated by Fig. 5. In Fig. 5 (a), both functional blocks FB1 and FB3 send data to functional block FB2. In Fig. 5 (b), packets sent by the functional block FB1 are redirected to FB4 while the data sent by FB3 continues to be sent to FB2. The header of the packets sent by FB1 still contains the same IDP value 'c'. There is no way to identify from its numerical value whether an IDP is public or private. However the IDT maintains two entries $<\mathtt{c}_* \rightarrow \mathtt{FB2}>$ and $<\mathtt{c}_{FB1} \rightarrow \mathtt{FB4}>$ and dispatches packets to the appropriate destination according to the sender FB.

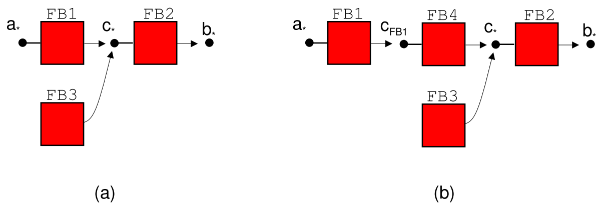


Fig. 5. Restricted IDP re-binding.

C. Compartments

To enable communication between hosts and routers implementing and using the same set of functional blocks, we also introduce the concept of *compartments*. A compartment is a set of FBs, IDPs and ICs with some commonly agreed set of communication principles, protocols and policies. Typical network compartments are an Ethernet segment, the public IPv4 Internet, a private IPv4 subnet, the DNS, peer-to-peer systems like Skype, or distributed web caching networks like Akamai. Note that the concept of compartment is different to the notion of *layer*, in the sense that the compartment concept captures the idea of a *network instance* regardless of the level(s) at which it operates in the network architecture.

In addition to network compartments, our architecture introduces a special compartment called the *node compartment*. We indeed consider each networking host to be itself a network composed by the functional blocks running on the host. The node compartment thus encompasses all FBs and IDPs within a node. Throughout this paper network compartments are depicted as dashed blue lines and the node compartment as a solid blue line.

III. Communication API

Network compartments are free to *internally* use whatever addressing, naming, routing, networking mechanisms, protocols, packet formats, etc., they want. The objective is to enable variability and evolution by not imposing any restriction on the internal network operation of compartments. However, to support all possible and unforeseen interactions, all compartments must support a generic "compartment API" which provides the "glue" that permits to build complex network stacks and packet processing paths.

A. Genericity Requirements

With respect to application developers, a key requirement is that the API primitives support generic arguments such that applications can use the same programming primitives to communicate with *any* compartment. In other words, the objective is that developers write "network-agnostic" applications that do not need to be modified to support newly developed network compartments. For example, one could write a "content browser" (similar to today's web browsers) that takes any *opaque* string as user input. The browser would then find the right network compartment for handling this address or name and it would require from it a communication channel to the destination. Eventually the browser would send the data to the target FB at the destination without, at any time, having to understand the syntax of the input string.

B. *User Provider Model*

We borrow a fundamental networking concept from the Open Systems Interconnections (OSI) [11] architecture which introduces the notions of user and provider layers. Basically, in OSI a provider layer N provides network services to all the N+1 layers of the architecture. For example, a network layer offers a connectivity service to all the transport layers.

We embed this fundamental networking concept into the architecture via the compartment API. Basically, a user functional block requests services from a provider layer via the compartment API: because it is generic, the user-provider relationship between compartments is not fixed and network compartments are potentially combined in any arbitrary way. Note however, that we do not mandate OSI layers or a static layering as in OSI.

C. *The compartment API*

As a starting point towards generality, the compartment API currently offers five fundamental primitives detailed below with some simplified C-style function prototypes. Similar to Plutarch [6], the API follows a publish/resolve communication model in which a *service* is *published* within a certain compartment's *context*. A published service then is *resolved* and the obtained IDP can be used to *send* data to the resolved service. Beside resolution, one also may *lookup* a service to obtain further reachability information but not instantiate an information channel to the service.

- `IDP`$_p$ `publish(IDP`$_c$`, CONTEXT, SERVICE)`
- `int unpublish(IDP`$_c$`, IDP published, SERVICE)`
- `IDP`$_r$ `resolve(IDP`$_c$`, CONTEXT, SERVICE)`
- `void* lookup(IDP`$_c$`, CONTEXT, SERVICE)`
- `int send(IDP`$_r$`, DATA)`

In the primitives, IDP$_c$ identifies the compartment handling the request. The SERVICE argument is typically what is published or looked up, while the CONTEXT argument defines some scope inside the compartment. Note that the term *service* is used throughout this paper to refer to any kind of resource that is resolved. This, for example, includes addresses, names, network nodes, protocol entities, files, video streams, printing services, etc.

D. *Examples*

To further describe and illustrate the compartment API we present in the next sections some simple examples of compartment communications and describe how each scenario would be implemented with the basic primitives of the API.

1) Node Local Communication Setup: Since each node is organized as a compartment, the communications *inside* a node are also setup via the compartment API. In particular, functional blocks publish IDPs and resolve services inside the node compartment with the same primitives used to communicate with network compartments. As a basic functionality, this allows each functional block to discover the services and network compartments available inside the node in which it is running.

For example, a functional block implementing the Ethernet protocol publishes itself inside the node compartment with the following primitive:

```
y ← publish(NODE_IDP, ".", "ETHERNET")
```

Generally, NODE_IDP is provided by the node compartment. In this example CONTEXT is defined as `"."` which restricts all operations to the local node. Upon success, the publish primitive returns the IDP 'y' (randomly generated) which is now bound to the Ethernet functional block. Now, a second functional block which wants to send data through an Ethernet interface will be able to resolve this IDP with the following request:

```
y ← resolve(NODE_IDP, ".", "ETHERNET")
```

This functional block (e.g., an IP stack), now uses the IDP 'y' for communication with the Ethernet functional block. A possible communication between an Ethernet and an IP functional block is shown in the next two examples.

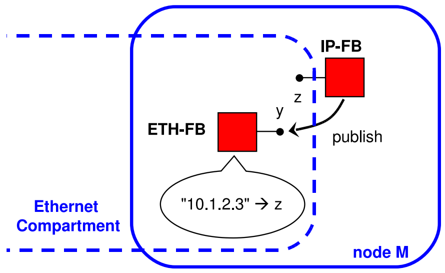


Fig. 6. The `publish` primitive.

2) Publishing an IP Address: In Fig. 6, the functional block implementing the IP stack (IP-FB) publishes its IP address inside the Ethernet compartment via the Ethernet functional block ETH-FB. This means that the IP-FB becomes reachable via the Ethernet compartment with the service name `"10.1.2.3"`. This publication is performed by the IP-FB by calling the primitive

```
z ← publish(y, "*", "10.1.2.3")
```

which, upon success, returns the IDP 'z'. Note that the generic CONTEXT `"*"` specifies the largest possible scope as interpreted by the compartment: for the Ethernet compartment this typically means all attached hosts on the segment and maps (inside the compartment) into the broadcast address `FF:FF:FF:FF:FF:FF`. For an IP compartment, the CONTEXT `"*"` would typically be interpreted as being the local subnet address (e.g., `"10.1.2.255"` for a class-C subnet). It is also possible to specify a particular CONTEXT: for example, with the Ethernet compartment, a multicast service may be published by specifying the Ethernet multicast address to be used (as shown on the following example):

```
m ← publish(y, "01:00:5e:00:00:09", "224.0.0.9")
```

3) Resolving an IP Address: Fig. 7 illustrates the `resolve` primitive. In this follow-up example, an “IP-stack” functional block inside Node N asks the Ethernet compartment (via ETH-FB) to resolve the service `"10.1.2.3"` in the `"*"` CONTEXT (i.e., the entire Ethernet segment). This is performed by calling the primitive

```
s ← resolve(e, "*", "10.1.2.3")
```

which, upon success, returns the IDP `'s'` which is used to send data to the resolved service. Inside Node N, the ETH-FB maintains some state that permits to transfer any data sent to the IDP `'s'` to the IDP `'z'` in Node M. For Ethernet, this information includes the MAC address of the destination node plus a dynamically allocated token value used for demultiplexing packets at the destination node.

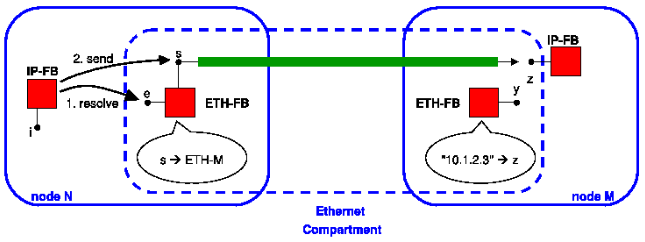


Fig. 7. The `resolve` primitive.

How the real resolution is performed is left to the compartment. In the case of the Ethernet compartment, a resolution request triggers the sending of a broadcast message (sent to `FF:FF:FF:FF:FF:FF`) containing the service name (here `"10.1.2.3"`) being looked up. Any Ethernet FB that has published the service in the `"*"` CONTEXT replies to the request with a message containing the information needed to reach the service via the corresponding compartment. For the Ethernet compartment this request-reply phase is actually similar to today’s operation of ARP (Address Resolution Protocol) but with dynamically assigned “type” values for the Ethernet header.

The `lookup` primitive is similar to the `resolve` primitive except that it does not instantiate an information channel to the service but solely returns reachability information about the service. In the case of the Ethernet compartment, a lookup may simply return a boolean answer indicating whether the service is reachable (and can be resolved) or not. The `lookup` primitive is actually useful for compartments that solely maintain mappings between multiple namespaces but do not forward data: this is for example the case of the Domain Name System which translates DNS names into IP addresses.

IV. USE CASES

A. Dynamic Reconfiguration

The rebinding property of IDPs is used to dynamically reconfigure the data path within a node. The following scenarios show the usefulness of this functionality.

1) Monitoring: Today, observing the network environment is of great interest, for example for Intrusion Detection Systems (IDS), traffic monitoring, etc. With the proposed architecture, the system logic may insert some monitoring functional block(s) at any position in any data path without disrupting the data flow. Adding an extra FB in the data path indeed only involves some basic IDP re-binding as was illustrated in Figures 2 and 3. If we assume that the added component FB3 performs some monitoring operation, this examples shows how easy it is to insert monitoring probes in our architecture.

2) Network Agnostic Applications: Today’s applications need to explicitly specify the protocols they are using, since they have to specify the socket type and address family. This complicates the introduction of new network protocols. However with the architecture proposed in this paper, applications do not have to understand anything about the underlaying network compartment. As a proof of concept we have written a chat application which runs on top of any network protocol (e.g., Ethernet, IP), without the need to change anything in the source code.

3) TCP over ETH: The IP layer is not needed in all network scenarios. In small scale networks, as Personal Area Networks or Sensor Networks, the IP layer is not mandatory or even procudes unnecessary overhead. With legacy systems, it is not possible to bypass the IP layer “on-demand”. With the help of the IDP concept, the communication path within a node is setup or re-configured such that it is optimized for any given networking scenario. With our architecture, it is possible to dynamically reconfigure the network stack and use transport layer protocols directly over Ethernet.

4) Others: There are many other scenarios in which the dynamic reconfiguration of IDPs is useful.

- Different functional blocks participate in the protocol stack depending on the current network environment. For example, upon switching from a wired network to a wireless network an encryption functional block may by added on the fly in all data paths.
- Upon congestion at the link layer, all the data could be compressed before it is actually sent over the link.
- Functional blocks with the same functionality are exchanged on the fly which allows for a system update at run time. For example, if one has updated a functional block that e.g., computes a checksum, the IDP that is

used to communicate with the original FB is bound to the new FB.

An additional use case is described in [12]. There, the authors describe how a *publish subscribe system* is implemented in the ANA framework.

B. Security Mechanisms

Beside the benefit of dynamic rebinding, IDPs are also used to enforce some security mechanisms.

1) Access Control: To enforce access control, a functional block may provide two kinds of IDPs. A public IDP that is used as an initial access point, to verify the credentials provided by the client functional blocks. After successful verification of the credentials a new private IDP is generated. This IDP may now only be used from the functional block that has successfully identified itself.

2) Default Off Policy: The concept of IDPs implement a *default off* policy [13]. For example, in the standard TCP/IP protocol stack, the number 6 in the "next protocol field" of the IP header, always refers to TCP. This static binding eases the flooding of the network with broadcast packets that have to be treated in the application layer of each node. With the use of IDPs this is not possible, since the IDPs differ from node to node. Therefore, before starting to send data, an information channel has always to be setup explicitly.

V. Discussion

One may argue that with every layer of indirection the system complexity increases and the performance gets worse. Indeed, there is an overhead introduced by the IDPs and during the communication setup. However the ultimate goal of our work is to provide a flexible network architecture which can be used to build an autonomic network architecture. We believe that this is not possible with the current "monolithic and statically bound" networking architectures and therefore the overhead introduced with the IDPs is justifiable. Additionally, our architecture is able to compose a protocol stack which is optimal for a given communication. For example, one could decide to entirely skip the IP protocol for LAN communications and run transport protocols directly over Ethernet. This means that we omit unnecessary "layers" and therefore increase the overall system performance. But, at the current project status, we are not able to present an evaluation which compares the flexibility gains with the performance cost.

Another interesting aspect of the architecture is to determine the conditions under which a redirect is as simple as described in this paper. We see difficulties in the transition of state: If a functional block should be exchanged with another functional block the internal state has to be transferred. Since this state is different for each functional block there is no generic way to transfer state between two functional blocks. However, we note that state migration is not a problem specific to our architecture and we plan to re-use mechanisms developed for other architectures to (at least partially) resolve that problem. For example, migration mechanisms related to TCP are presented in [14] or [15].

A third aspect that needs further investigation is related to the lifetime of IDPs. Currently there are two types of IDPs: permanent IDPs and volatile IDPs. Permanent IDPs are registered as long as a given functional block is loaded. Volatile IDPs disappear when they are not used anymore. All IDPs bound to an IC should be volatile to allow easy garbage collection when an IC is not used anymore or when the destination node has crashed. Whereas IDPs that belong to a functional block published in the node compartment are permanent and they will be available as long as the corresponding functional block is loaded.

VI. Related Work

A. Configurable Protocol Stacks

The idea of composing a protocol stack dedicated for a specific use is not new. It dates back to the x-Kernel [16] which defines an explicit architecture for constructing and composing network protocols or even to the UNIX System V STREAMS [17] implementation which allows data processing modules to be connected. A more recent project is the Click Modular Router [18] in which the networking functionality is divided into small elements each of which performs a simple computation such as decrementing a TTL field or queuing a packet. However none of these architectures are designed to change the protocol stack at run time. Several other projects are targeted exactly on the dynamic configuration of protocol stacks. In the Horus system [19], protocol layers can be stacked on top of each other at run time or the ASH [20] system even allows to generate code on-the-fly thereby allowing optimal memory handling. Finally, Protocol Boosters [21] allow to enhance a protocol stack on the fly with functions such as encryption or compression. In contrast to our architecture Protocol Boosters depend on an existing protocol stack to which they add additional networking functions dynamically.

Inspired by the flexibility of these dynamic protocol stacks we have built our own system. We take this flexibility to the next level by introducing network compartments. Each of these compartments may have its own protocol system thereby allowing different networking architectures to coexist. This goes clearly beyond the legacy TCP/IP protocol stack for which the other projects are targeted.

B. Network Architectures

In recent years the interest of researchers in network architectures has increased. As a result there are some projects which design a clean slate approach for the Internet. Most of those projects belong either to the US FIND [22] initiative or to the EU FIRE [23] initiative. Our architecture is one of these approaches which try to build a network which is better armed for the future. Our architecture is inspired by several other network architectures. Similar to Plutarch [6] we have explicitly designed our architecture to allow a heterogeneous network environment. In Plutarch the network is divided into contexts, within one context the network is assumed to be homogeneous. This is similar to the notion of compartments in ANA. However, Plutarch only specifies a straw man API whereas we have a running implementation. Another work that influenced our architecture are Network Pointers [9]. A network pointer is a packet processing function that is addressed with a pointer value. This is similar to our notion of information dispatch points, however network pointers are exchanged between different nodes, whereas IDPs are only a node local concept.

VII. Conclusions

In this paper, we have presented the ANA framework that is based on the principle of indirection. We have successively improved our architecture during several implementation and design cycles, and continue to populate it with more functionality.

We have contributed multiple new elements to the existing network architectures. We have provided abstractions and concepts that describe the interactions between the network functions within one node or between nodes. Specifically, we have introduced the concept of information dispatch points that enables the decoupling of sending and receiving functionality. This in turn allows new communication principles to arise. In contrast to existing systems, with our framework it is possible to rearrange the protocol stack on a node at run time.

The distinction between public and private IDPs offers an elegant mean to implement access control mechanisms limiting the availability of network functions or protocols. This functionality is an important cornerstone for future security mechanisms.

With the definition of the communication API, we have provided a generic interface for the interaction between node internal functionality as well as for the communication between network nodes. The objective of this API is to enable flexibility without imposing restrictions on the implementation of any functionality.

To assess the suitability of the API, it is best to apply it to existing protocols. In our ongoing work, we have already demonstrated that the API captures the interactions of existing protocols, such as for example Ethernet, IP, or DNS.

As a next step, we plan to add more formalism to the work described in this paper. A possible future direction is to link the architectural work and its abstractions with the ABC work described in [24].

We believe that the advantages of the proposed architecture are also best demonstrated by further extending the current implementation. Ultimately, the users of the framework will adjudicate upon its success.

VIII. Acknowledgment

This work was performed within the ANA (Autonomic Network Architecture) project [25] which is funded by the European Union Information Society Technologies Framework Programme 6. The aim of the ANA project is to provide a framework to flexibly host, interconnect, and federate multiple heterogeneous networks in an autonomic way, i.e., without requiring active human intervention.

References

[1] D. Clark, "The design philosophy of the darpa internet protocols," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*. New York, NY, USA: ACM, 1988, pp. 106–114.

[2] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In vini veritas: realistic and controlled network experimentation," in *Proceedings of SIGCOMM 2006*. New York, NY, USA: ACM, 2006, pp. 3–14.

[3] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible ip router software," in *Proceedings of NSDI 2005*. Berkeley, CA, USA: USENIX Association, 2005, pp. 189–202.

[4] Z. Turanyi, A. Valko, and A. Campbell, "4+4: An Architecture for Evolving the Internet Address Space Back Toward Transparency," *Computer Communication Review*, vol. 33, no. 5, pp. 43–54, October 2003.

[5] D. Clark, R. Braden, A. Falk, and V. Pingali, "FARA: Reorganizing the Addressing Architecture," in *Proceedings of ACM SIGCOMM FDNA Workshop*, August 2003, Karlsruhe, Germany.

[6] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield, "Plutarch: an argument for network pluralism," in *Proceedings of ACM SIGCOMM FDNA Workshop*, August 2003, Karlsruhe, Germany.

[7] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 73–86, 2002.

[8] C. Jelger, C. Tschudin, S. Schmid, and G. Leduc, "Basic Abstractions for an Autonomic Network Architecture," in *Proceedings of AOC'07*, June 2007, Helsinki, Finland.

[9] C. Tschudin and R. Gold, "Network Pointers," in *Proceedings of the ACM HotNets-I Workshop*, October 2002, Princeton, NJ, USA.

[10] S. Ganapathy and T. Wolf, "Design of a network service architecture," in *Proceedings of ICCCN '07*, Honolulu, HI, Aug. 2007.

[11] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, April 1980.

[12] T. Hossmann, A. Keller, M. May, and S. Dudler, "Implementing the future internet: A case study of pub/sub in ANA," in *Proceedings of CFI '08*, Seoul, Korea, 2008.

[13] H. Ballani and Y. Chawathe and S. Ratnasamy and T. Roscoe and S. Shenker, "Off by Default!" in *Proceedings of the ACM HotNets-IV Workshop)*, November 2005.

[14] F. Sultan, K. Srinivasan, and L. Iftode, "Transport layer support for highly-available network services," in *Proceedings of HOTOS '01. Extended version: Technical Report DCS-TR-429, Rutgers University*. Washington, DC, USA: IEEE Computer Society, 2001, p. 182.

[15] S. M. ElRakabawy, A. Klemm, and C. Lindemann, "Gateway adaptive pacing for tcp across multihop wireless networks and the internet," in *Proceedings of MSWiM '06*. New York, USA: ACM, 2006, p. 173.

[16] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Trans. Softw. Eng.*, vol. 17, no. 1, pp. 64–76, 1991.

[17] D. M. Ritchie, "A stream input-output system," *UNIX Vol. II: research system (10th ed.)*, pp. 503–511, 1990.

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

[19] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr, "A framework for protocol composition in horus," in *PODC '95: Proceedings of the fourteenth annual ACM symposium on PODC*. New York, NY, USA: ACM, 1995, pp. 80–89.

[20] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "Ashs: Application-specific handlers for high-performance messaging," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 4, pp. 40–52, 1996.

[21] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, and T. Raleigh, "Protocol boosters," *Special Issue on Protocol Architectures for 21st Century Applications*, vol. 16, no. 3, pp. 437–444, 1998.

[22] "FIND – Future Internet Design (FIND) - US National Science Foundation," at http://www.nsf.gov/pubs/2006/nsf06516/nsf06516.htm.

[23] "Future Internet Research and Experimentation (FIRE) initiative - European Commission," at http://cordis.europa.eu/ist/fet/comms-fire.htm.

[24] M. Karsten, S. Keshav, S. Prasad, and M. Beg, "An axiomatic basis for communication," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 217–228, 2007.

[25] "Autonomic Network Architecture - EU Project (2006-2009)," http://www.ana-project.org.