AUTONOMIC CONFIGURATION OF DYNAMIC PROTOCOL STACKS

Ariane Keller, Stephan Neuhaus, Markus Happe [*]

Communication Systems Group
ETH Zurich, Zurich, Switzerland
email: first.last@tik.ee.ethz.ch

Daniel Borkmann [†]

Red Hat
Zurich, Switzerland
email: borkmann@redhat.com

ABSTRACT

The Internet architecture works well for a wide variety of communication scenarios. However, communication in constrained environments with embedded and/or mobile devices requires specialized communication protocols. Additionally, network characteristics often vary in those scenarios, which makes it difficult for a static set of protocols to provide the required functionality. Therefore, we propose a self-aware configuration method for dynamic protocol stacks that allows for the autonomic configuration of individual protocols into a protocol stack. This adaptation happens at run-time and might be triggered by policy changes or by changing network conditions. We demonstrate the efficiency of our self-aware architecture for a networking scenario where the link quality changes over time. In contrast to a static reliable protocol stack we can reduce the communication overhead in terms of sent packets by 28% for a given scenario.

1. INTRODUCTION

In contrast to the beginning of the computing age, nowadays most applications are distributed and interact with other devices. Today's applications are executed on a variety of devices (such as workstations, notebooks, cellphones, sensor nodes) with different processing power and in varying network conditions (such as wireless or wired, trusted or untrusted, etc.). We can no longer assume that a static networking architecture always provides robust and secure communication links with high throughput at low performance overhead and power consumption.

For instance, mobile devices are usually used in dynamic network environments, where the link quality can vary dramatically over time, e.g., when a user moves away from or approaches a WLAN hotspot. Moreover, the user may switch between private and public networks, which may require different privacy modes. Static networking architectures usually lack the flexibility to adapt themselves to dynamic environments in order provide the required communication functionalities at minimal cost.

In the current Internet architecture certain protocols can already adapt themselves to changing communication conditions. However, the overall functionality to be provided by a communication link has to be specified while writing an application and can only be selected from a small pre-defined range. We argue that in order to execute applications optimally in dynamic network environments, we need a self-aware communication architecture, in which the overall functionality autonomously adapts itself to the current network characteristics. Examples of such an adaptation could be the dynamic inclusion of a reliability and/or a privacy block in the protocol stack whenever the network conditions demand them.

In previous work [1, 2] we have already proposed to use dynamic protocol stacks instead of static protocol stacks. Dynamic protocol stacks (DPS) split the networking functionalities into individual functional blocks, which can be dynamically linked with each other in order to form arbitrary protocol stacks. In this paper we extend our work by introducing a self-aware networking architecture that adapts the protocol stacks at run-time to a changing environment.

Specifically, our contributions are:

- We have developed a self-aware network node architecture that supports the autonomic configuration of dynamic protocol stacks.
- We have developed techniques to set up and adapt protocol stacks based on application requirements and the current network condition.
- We have evaluated our self-aware architecture with a real-world scenario and shown that self-adaptation of the protocol stack can reduce the communication overhead in terms of sent packets as compared to static stacks.

The rest of this paper is structured as follows: We first give an overview of related work (Section 2). Then, we present our self-aware networking architecture and our self-adaptation strategies in Section 3. Next, we demonstrate the efficiency of our approach in a real-world networking sce-

[*]The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n° 257906.

[†]This work was performed while affiliated with ETH Zurich.

nario with changing link qualities (Section 4). Finally, Section 5 concludes the paper.

2. RELATED WORK

Already in the early 1990s, Tennenhouse and Wetherall proposed *Active Networks* in which users could inject custom code into the network [3]. This code was associated with a set of packets that traversed the network from the source over several routers to the destination. The code was executed on intermediate nodes and could modify the packets on-the-fly as desired. Less flexible architectures were proposed by the `Click` modular router [4] and `netgraph` [5]. Both `Click` and `netgraph` offer the possibility to combine networking functionalities flexibly. However, they did not focus on run-time reconfiguration. The concepts of *flexibility, modularity, and extensibility* were also recently presented by Ghodsi et al. [6] as the basic requirements for a network architecture that is able to evolve. Wolf et al. [7] argue that a user should be able to *choose* the service that best fits his requirements. In contrast to related work we present a novel self-aware networking architecture which adapts its protocol stack autonomously to react to a changing environment without fine-granular user interaction.

3. SELF-AWARE NETWORKING ARCHITECTURE

In this section we describe our self-aware networking architecture that enables us to dynamically configure protocol stacks. We first discuss the architecture developed for the node-local adaptation and then we focus on the setup and adaptation of the protocol stack between nodes.

3.1. Node-Local Adaptation

Figure 1 shows our self-aware network node architecture, which consists of the following building blocks:

- The **network models** contain the network protocols, the network characteristics, as well as some predictions on how the world might look like in the future.
- The **sensors** provide information such as the signal-to-noise ratio, available energy, or throughput. Sensors can be passive (just observing) or active (inserting probes in the network, observing the reaction).
- The **sensor daemon** collects data from the individual sensors. It offers additional functionality such as sending notifications whenever a monitored value exceeds a specified threshold.
- The **self-adaptation engine** contains a *strategy finder*, which selects the current strategy (minimize power, maximize throughput, etc.), and a *stack builder*, which determines the best stack and adapts the networking core accordingly.
- The **networking core** is responsible for processing network packets. Therefore, it passes a network packet between functional blocks. The details of our networking core are described in [2].

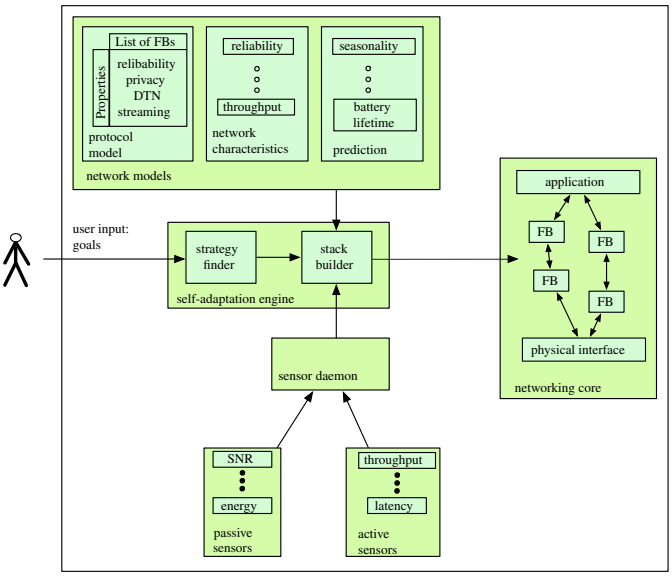

Fig. 1. Overview of the self-aware node architecture.

Building a protocol stack requires the knowledge of (a) the available protocols and (b) the communication requirements. In the Internet architecture, an application solves this problem by using a specific BSD socket type and additional libraries as needed, e.g., for encryption. This setup implies that once the application is written it will always use the same protocols and it cannot make use of newly developed protocols that might fit its needs just as well or even better. In our self-aware networking architecture, the application can specify a set of properties that need to be fulfilled for a given communication. The stack builder then examines the protocol models and finds all protocol stacks that match the requirements. In the current implementation, both protocols and requirements, are specified with simple key words.

3.2. Inter-Node Adaptation

Once all possible stacks are known, a connection to the destination node has to be established. The destination node might not have the required protocols available; therefore, before the communication starts, a protocol stack negotiation phase is executed. First, all possible protocol stacks are sent to the destination node. The destination node decides which protocol stack to use, sets up this protocol stack and sends the chosen configuration back to the source. If the source never receives a reply from the destination, which could happen on a lossy link, the source re-sends the con-

figuration message and waits for the confirmation. After the completion of the negotiation phase, the actual data transmission starts. In order to solve the "chicken and egg problem" of the protocol used for the protocol negotiation phase, we assume that all nodes in a given network segment use the Ethernet protocol. Similarly, if a connection to a node in another segment should be established, the intermediate nodes have to use the same routing protocol.

Upon receiving a data packet, a node has to decide how to process it. In the Internet architecture this decision is based on *next header fields* that are part of each protocol header. For example, in the next header field of the Ethernet protocol it is specified whether the next protocol is `IPv4`, `IPv6`, `ARP`, etc. If the protocol stack is negotiated upfront, this step by step resolution of the next protocol is not necessary, instead, a single identifier per connection can be used. This identifier is calculated by the stack builder as follows: Every functional block has a unique name. In order to obtain a unique name the inverted `url` that is associated with the developer is used. This is similar to the convention for package names in the Java programming language. The unique identifier for the overall protocol stack is then obtained by concatenating the individual names and hashing them. If the identical protocol is implemented by several developers, and their implementations pass an interoperability test, a special interoperability name should be used. Upon packet reception, the Ethernet functional block checks the hash and forwards the packet to the corresponding stack "pipeline".

When the networking conditions change, the self-aware nodes might want to change the protocol stack to add or remove networking functionalities. Identifying a given stack by a unique identifier is also valuable when changing the protocol stack on-the-fly. The negotiation of the new protocol is similar to the negotiation for setting up a protocol. The re-negotiation is executed over the currently used protocol. While adapting the protocol stack packets might be reordered on their way from source to destination. Therefore, special care has to be taken that packets still belonging to the old stack are not processed by the new stack and vice versa. Since the hash that identifies a given stack will change when the protocol stack is changed, also the packets sent over the new stack will be identified with a different hash. This hash is used to dispatch the packet either to the new or the old protocol stack. Figure 2 depicts this change of the protocol stack.

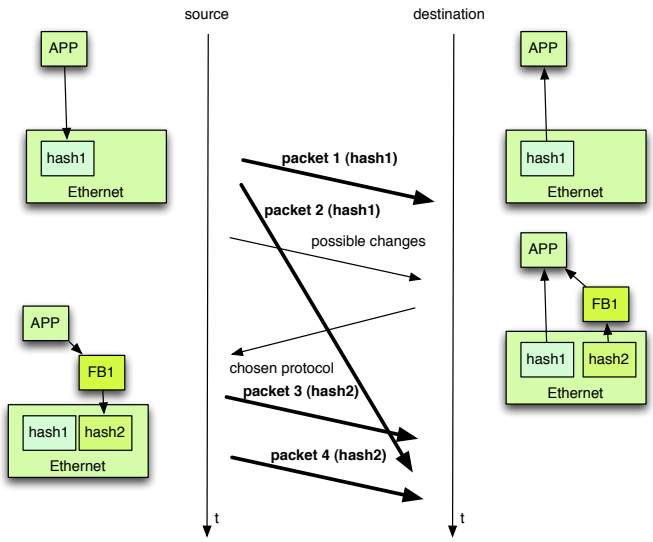

Fig. 2. Updating the dynamic protocol stack over time.

4. EXPERIMENTAL RESULTS

We implemented the self-aware network node architecture as a combination of Linux kernel modules (for the networking core) and user-space tools (for monitoring and configuration). However, it could be implemented on any other operating system as well. Our implementation is designed to scale from small embedded systems up to high-end SMP servers. Applications interact with the network architecture over a new BSD socket family that supports the following socket calls: *open, ioctl, sendto, poll, recvfrom, close*. A library is provided that allows for specifying the communication requirements. The source code of our architecture together with getting started information is available in github at `http://github.org/epics/reconos`.

In order to evaluate the benefits of a self-aware network architecture, we show how our system autonomously adapts itself to changing network conditions. We developed a simple application that mimics a sensor that sends measurement data periodically to a server. We argue that transmitting a packet over a wireless interface costs energy, and therefore should only be performed when necessary. Therefore, we implemented a stack builder that includes an idle repeat request (IRR) reliability protocol in the protocol stack, only when sensors report low link quality. The link quality is determined by a sensor that divides the current with the maximum possible wireless link quality. Our link-quality-aware networking architecture is shown in Figure 3.

We evaluated our architecture on commodity notebooks. In order to obtain reproducible results, we used a wired connection between the test machines and used the Linux traffic control tool `tc` with the `netem discipline` [8] to emulate packet loss. We recorded the link quality between two nodes while walking around in our office building, see Figure 4. We have used this recording as realistic input for our emulation. Simultaneously, we measured that packets got lost, when the link quality was below 35%.

Our stack builder requests to be notified by the sensor daemon when the signal strength falls below a threshold of 40% or increases beyond 50%, see Figure 4. Upon such an

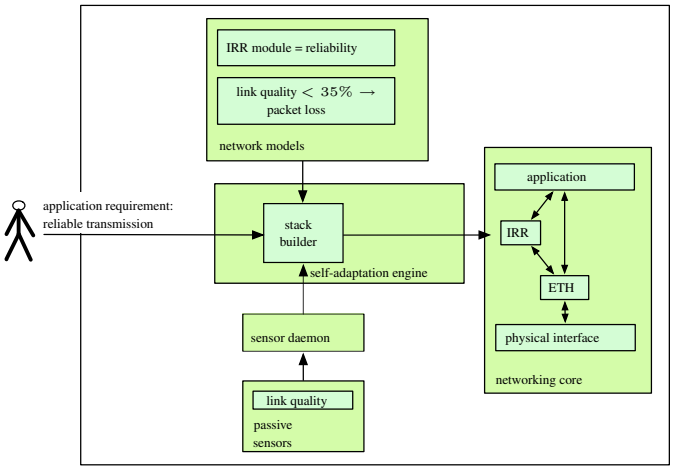

Fig. 3. Implementation of the node architecture for the link-quality-aware protocol stack.

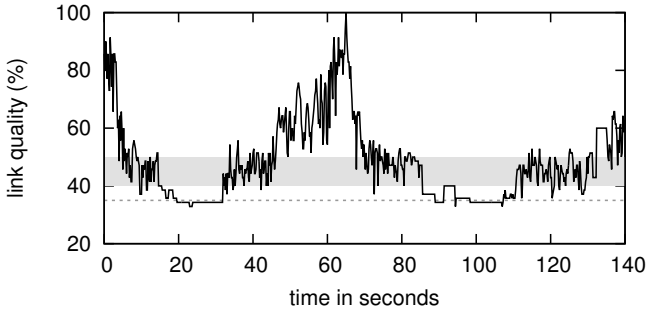

Fig. 4. Measured link quality over 140 seconds. Packets got lost when the link quality was below the dashed line. The DPS is updated when the graph crosses the gray bar.

event, it either inserts the reliability module or it removes the reliability module, and renegotiates the protocol stack with the neighboring node. The lower threshold for renegotiation ensures that the reliability protocol is inserted to the protocol stack before the link quality reaches the critical value of 35%. The upper threshold is used to avoid frequent adaptations of the protocol stack.

For evaluation purposes, we compared the data loss rate and the total number of packets sent for (i) a protocol stack that dynamically adapts itself to the link quality, (ii) a protocol stack that never uses reliability, and (iii) a protocol stack that always uses reliability. We used these measured values to emulate the network conditions on a machine that connected the two test machines.

Table 1 summarizes our results. The configuration with no reliability lost on average 31% of the packets, whereas we didn't observe packet loss in the other two configurations. However, this reliability comes at a price. The overhead (in terms of sent packets) for achieving reliability was 128% for the configuration that was statically configured to use the reliability protocol. The total overhead for the dynamic configuration was 100% split in 60% for sending acknowledgement and retransmission packets and 40% for sending the protocol stack reconfiguration messages. This clearly shows that adaptive protocol stacks can reduce the total communication overhead in dynamic scenarios. However, the adaptation algorithm has to be designed carefully to avoid increasing the total overhead by sending too many stack reconfiguration messages.

Table 1. Comparison between static and autonomous configurations over 140 seconds.

		overhead	
config.	packet loss rate	reliability	reconfig.
unreliable	31%	-	-
reliable	0%	128%	-
autonomous	0%	60%	40%

We also measured the protocol stack reconfiguration time, i.e., the time it takes from an event that triggers a reconfiguration until data can be sent over the new protocol stack. This time is composed of (i) the time to determine and reconfigure the stack on both sides of the communication and (ii) the time to send the reconfiguration messages. We measured a protocol stack reconfiguration time of 806μs whereof 286μs were required for the transmission of the packets (round trip time).

5. CONCLUSION

In this paper we presented a novel self-aware network node architecture. The self-adaptation of the protocol stack is triggered by combining the sensor input with models and goals. The currently implemented self-adaptation techniques allow to insert or remove protocols, such as encryption or reliability, at run-time. We demonstrated that our self-aware networking architecture can autonomously adapt its protocol stack to varying link qualities without loosing any packets while reducing the communication overhead (in terms of sent packets) by 28% compared to a static networking architecture.

In future work we will focus on more advanced self-adaptation algorithms and we will apply our architecture to a smart camera network. The platform for the smart cameras will be ReconOS [9], which allows us to execute some parts of the network functionality, such as encryption or compression algorithms, in hardware, while still being able to freely compose and adapt the protocol stack.

6. REFERENCES

[1] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ana)," *Selected Areas in Communications, IEEE Journal on*, vol. 28, no. 1, pp. 4–14, Jan. 2010.

[2] A. Keller, D. Borkmann, and W. Mühlbauer, "Efficient implementation of dynamic protocol stacks (poster)," in *Proc. ACM/IEEE Symp. on Architecture for Networking and Communications Systems (ANCS)*. Washington, DC, USA: IEEE Computer Society, Oct. 2011, pp. 83–84.

[3] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *Computer Communication Review*, vol. 26, pp. 5–18, 1996.

[4] 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.

[5] "netgraph – graph based kernel networking subsystem," (accessed in Sept. 2012). [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=netgraph\&sektion=4

[6] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, "Intelligent design enables architectural evolution," in *Proc. of the ACM Workshop on Hot Topics in Networks*, ser. HotNets-X. NY, USA: ACM, 2011, pp. 3:1–3:6.

[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," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 105–106, Aug. 2012.

[8] "netem," (accessed June 2013). [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[9] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 8:1–8:33, Oct. 2009.