

100 Gbit/s End-to-End Communication: Adding Flexibility with Protocol Templates

Steffen Büchner*, Alireza Hasani[†], Lukasz Lopacinski[‡], Rolf Kraemer[‡], Jörg Nolte*

Distributed Systems/Operating Systems Group*, Systems Group[†]

Brandenburg University of Technology Cottbus–Senftenberg
Cottbus, Germany

Email: {steffen.buechner*, hasani[†], joerg.nolte*}@b-tu.de
System Design[‡]

IHP

Frankfurt/Oder, Germany

Email: {lopacinski[‡], kraemer[‡]}@ihp-microelectronics.com

Abstract—High-speed protocol processing that provides data-rates of 100 Gbit/s and beyond to the application stresses the whole communication system up to its outer limits. Such a system can only be utilized by employing highly specialized, application specific protocols, that are tailored for certain communication parameters, such as the packet loss rate. However, the requirements for most applications are not static, and a protocol designer cannot anticipate all possible communication conditions upfront. The contradiction between specialized protocols and unknown communication parameters can be solved by adapting the protocol implementation on demand to the current communication conditions. However, such an approach needs a protocol description language that allows the automatic specialization of protocols. In this paper, we present the Protocol Engine Template Language (PETL), that allows the automatic implementation of protocols by a constructive approach for a variety of communication conditions from protocol implementation templates.

Index Terms—Protocol Template; 100 Gbit/s end-to-end; End2End100; Soft Real-Time Protocol Stream Processing

I. INTRODUCTION

Fast wireless communication is a key requirement for many of today's digital services and the expected data rates are continuously rising. The demand for higher data rates is widely acknowledged and research groups all over the world focus on new transmission technologies. However, the benefits of faster transmission technologies will trickle away without new protocols and new protocol processing paradigms that can efficiently utilize these data rates. One main challenge is that the end-to-end (from sender's host memory to receiver's host memory) protocol processing for ultra-high data rates of 100 Gbit/s and beyond stresses the whole communication system, i.e., the host, and the Network-Interface-Card (NIC), to the outermost. The implication is that every part of the communication system can easily become a bottleneck.

These challenges are addressed by the project End2End100¹. The project End2End100 is part of the DFG (German Research Foundation) priority program "100 Gbit/s

Wireless And Beyond"². While the priority program's target is the development of the necessary wireless transmission technologies for 100 Gbit/s wireless communication, our focus is set on a soft real-time protocol stream processing approach [1]. Seeing protocol processing from a stream-processing point of view enables the protocol developer to implement protocols so that they can be processed efficiently on a given hardware setup. To be able to utilize the provided theoretical data rate, these protocols and their implementation are designed for specific applications and specific communication parameters. Given that the communication conditions and requirements in communication systems are not static, the applicability of the designed protocol is limited, because the requirements can change due to the application, e.g., the data rate or latency requirements change, and the conditions can change due to the communication medium, e.g., the channel quality can vary. These changes can happen during an ongoing transmission and they can be spontaneous, short-term, long-term, alternating, or a combination of these.

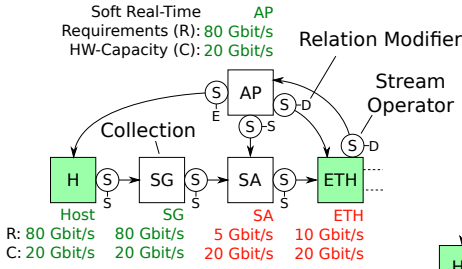
By manually handling special cases in the protocol implementation, side effects render the protocol design process cumbersome and error-prone. In this paper, we introduce the basics of a graph based description language that separates the protocol implementation from the special case handling. The Processing Engine Template Language (PETL) describes protocol implementations as templates, composed of pre-defined protocol building blocks, that can be instantiated into specific protocols. The handling of varying communication conditions is then covered by deriving a protocol implementation that suits the communication conditions on demand.

The remainder of this paper is organized as follows: Section II briefly introduces the basics of the PETL and shows its feasibility with an example. In section III we evaluate our approach with a scenario that simulates decreasing channel quality. The paper finishes with the discussion of related work and a conclusion.

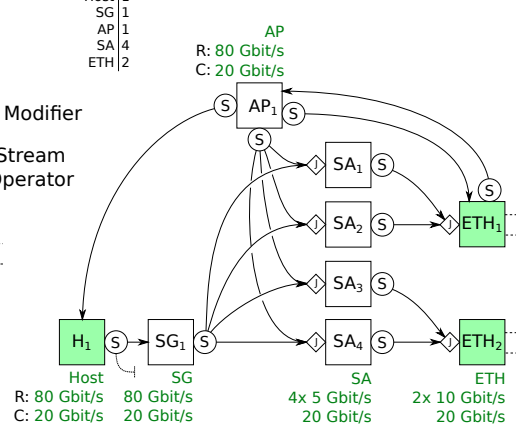
¹German Research Foundation Project End2End100, DFG NO 625/9-1, a joint project of IHP (Innovations for High Performance Microelectronics GmbH) and the Brandenburg University of Technology Cottbus–Senftenberg

²DFG Schwerpunktprogramm SPP 1655 Drahtlose Ultrahochgeschwindigkeitskommunikation für den mobilen Internetzugriff

a) Protocol Template



b) Collection | Count



c) Collection | Count

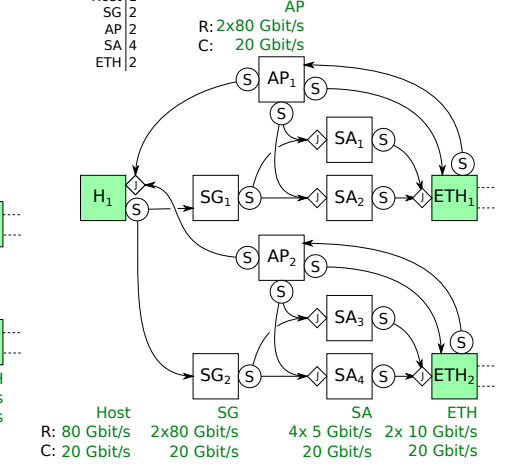


Fig. 1: a) The protocol template for a wireless data-link protocol. b) The specialized template for a data rate of 20 Gbit/s. c) A specialization that utilizes two full protocol pipelines to gain robustness against packet loss on one channel.

II. PROTOCOL TEMPLATES FOR PROTOCOL IMPLEMENTATIONS

The protocol implementation process presented in [1] results in a Protocol Processing Engine (processing engine) that connects Protocol Processing Stages (stages) by data streams that are manipulated by Stream Operators (stream operators). Such a processing engine is described as a graph, i.e., each node describes a stage (a protocol task's type and configuration), and an edge describes an individual data stream between two stages. That graph can be augmented with the needed stream operators and the calculated parallelization ratios. By means of that approach, we can use one processing engine description for a variety of adaptations, i.e., we can provide a tailored protocol implementation for different data rates and communication scenarios [1] from a single description that acts as a template for a certain class of protocols.

The PETL uses *collections* and *relations* as primitives. A collection describes a parallelized set of stages of the same type, as calculated during the analyzing phase of the design process. The collection is represented by a node that is augmented by the adaptation count, its type and name, as well as the configuration for the individual stages and which stages are grouped, i.e., share a configuration.

The data streams between the stages in a processing engine are represented by the *relation* primitive of the PETL. Like the collections, which describe a set of stages, a relation describes a set of data streams between two collections. The relation depends on the stream operator that is used to connect two collections, e.g., Stream-Split, Stream-Join, or Stream-Duplicate, and one of the following Relation Modifiers (modifiers) that states how a relation will unfold.

symmetrical – The source and destination stages are divided into symmetrical subsets for which the output's stream operator is applied individually.

Example: The relation $SG \rightarrow SA$ (Fig. 1a) unfolds into the two subsets $SG_1 \rightarrow (SA_1, SA_2)$ and $SG_2 \rightarrow (SA_3, SA_4)$ (Fig. 1c).

all-to-all – All source stages are connected with all destination stages.

direct – The stages are connected pairwise. Some stages will be unconnected in the case that the number of source and destination stages differ.

Example: The relation $AP \rightarrow ETH$ (Fig. 1a) unfolds into the single direct connection $AP_1 \rightarrow (ETH_1)$, ETH_2 stays unconnected (Fig. 1a). The same relation unfold into two direct connections $AP_1 \rightarrow (ETH_1)$ and $AP_2 \rightarrow (ETH_2)$ (Fig. 1b).

explicit – The relation is connects a source and destination stage explicitly.

Example: The relation $AP \rightarrow H$ (Fig. 1a) connects each AP stage explicitly with the H stage (Fig. 1b/c)

In the following, the feasibility of the PETL will be demonstrated with the protocol shown in figure 1a). This protocol is a variation of the data link protocol presented in [1]. The data link protocol is designed for 100 Gbit/s wireless communication and is based on 16 MB data chunks. A data chunk itself is divided into Sub-Packets of which each consists of the data, as well as redundancy for error correction. The sub-packets, which are smaller than a network frame, are aggregated into the larger network frames. That way it is possible to partially retransmit not only data chunks but also parts of network frames, reducing the data loss in the case an unstable communication channel is used. For more details of the protocol itself can be found in [1] and [2].

The presented protocol becomes a protocol implementation template by augmenting the data streams between the individual processing stages with relations. The protocol template of the sender side is shown in figure 1 a). The template consists of the 5 collections: Host (H), Sub-Packet Generator (SG), Sub-Packet Aggregator (SA), Ack-Processor (AP) and the physical Ethernet (ETH) interface. These collections are connected by seven relations. Such a template can be specialized by defining each collection's configuration, e.g., the processor core mapping, and the count attribute of the collection. The count attribute defines how many processing stages of the same type work in parallel. In most cases, this is determined

by the desired data-rate and the processing capacities of the underlying hardware.

In figure 1a) the template is augmented with the hardware's processing capacities and the soft real-time requirements for the desired data-rate (both in Gbit/s). The target data-rate in this example shall be 20 Gbit/s. The ratio (rounded up) of the hardware's capacity and the soft real-time requirements is the minimum count for each collection so that the desired data-rate can be achieved (Details on measuring the hardware's capacities and deriving the soft real-time requirements can be found in [1]).

The parallelization ratios lead to the following collection configuration: $1 \times \text{Host}$, $1 \times \text{SG}$, $1 \times \text{AP}$, $4 \times \text{SA}$ and $2 \times \text{ETH}$, that unfold the relations into the protocol implementation shown in figure 1b). The unfolding is explained with the following relations as an example: Host (H) $\rightarrow$ Sub-Packet Generator (SG), Sub-Packet Generator (SG) $\rightarrow$ Sub-Packet Aggregator (SA), Sub-Packet Aggregator (SA) $\rightarrow$ Ethernet (ETH) and Ethernet (H) $\rightarrow$ Ack-Processor (AP).

Since there is only one Host (H) stage and one Sub-Packet Generator (SG) stage, the symmetric relation connects both stages directly, i.e., the stage SG_1 is connected with the first slot of the output and the second slot stays unconnected. The symmetric relation between the Sub-Packet Generator (SG) and the Sub-Packet Aggregator (AP) unfolds into the two subsets ($\text{SA}_1, \text{SA}_2 \rightarrow \text{ETH}_1$) and ($\text{SA}_3, \text{SA}_4 \rightarrow \text{ETH}_2$). Since the number of elements in the source subset is higher than the number of elements in the destination subset, the outputs of the source stages are joined into the ETH stages. The direct relation between Ethernet collection (ETH) and Ack-Processor collection (AP), unfolds into a single connection, that connects the AP_1 stage with the ETH_1 stage, while the stage ETH_2 stays unconnected. This specialization is similar to the low latency scenario presented in [1].

Another specialization in which the count attributes of the Sub-Packet Generator (SG) and Ack-Processor (AP) are changed to 2 is shown in figure 1c). This protocol template specialization results in two complete protocol pipelines, whose source is the Host's output. The Symmetrical modifier connects the second Sub-Packet Aggregator (SG_2) with the second slot of the output, allowing the Host (H_1) to select one of the protocol pipelines for the transmission of a data-chunk. The symmetric modifier between Sub-Packet Generator (SG) collection and the Sub-Packet Aggregator (AP) collection generates the two subsets ($\text{SG}_1 \rightarrow \text{SA}_1, \text{SA}_2$) and ($\text{SG}_2 \rightarrow \text{SA}_3, \text{SA}_4$), which is then symmetrically joined with the ETH collection.

The relation between the Ethernet (ETH) collection and the Ack-Processor (AP) collection should connect each Ack-Processor with an individual Ethernet interface, and is, therefore, an example use of the direct modifier. Due to the direct modifier, the second Ack-Processor (AP_2) is connected to the second Ethernet interface (ETH_2), i.e., each Ack-Processor uses an individual ETH stage for its acknowledgments. Both of the specified protocol pipelines transmit data-chunks independently, e.g., they process individual acknowledgments, this

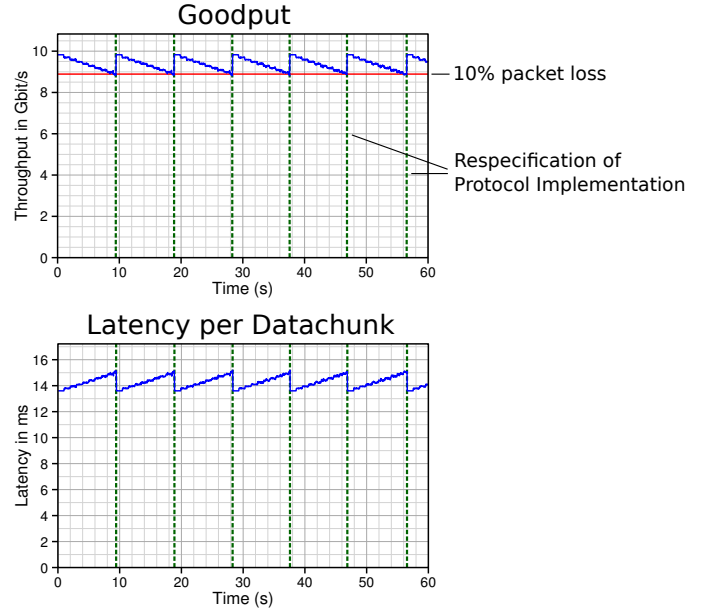


Fig. 2: The channels bit error rate is continuously increasing. Whenever a packet loss rate of 10% (horizontal line) is reached, the communication physical interface is switched (green line) by re-specifying the protocol template.

is similar to the stable throughput scenario presented in [1].

III. EVALUATION

The evaluation of the presented protocol template approach was conducted by using two Mellanox TilEncore Gx72 [3] embedded many-core boards. Each board is equipped with 72 processing cores with a clock frequency of 1 GHz, used for the protocol processing, four memory controllers (each 4 GB Memory), and eight 10 GbE interfaces. The TilEncore Gx72 boards are directly connected with each other using the 10 GbE interfaces, providing a theoretical maximum throughput of 80 Gbit/s when ignoring any protocol overhead. Since our evaluation setup is not able to provide a goodput, i.e., the actual application level throughput, of 80 Gbit/s from the host system to the TilEncore Gx72 many-core boards, the actual copying of bulk data from the host to the TilEncore boards was omitted. However, the data chunks are considered valid for the protocol processing, therefore the evaluation results are obtained by using a fully functional communication system that is controlled and set up by the hosts, only without copying bulk data between the host and the TilEncore boards.

In this evaluation scenario, the sender tries to send data with a constant data rate of 10 Gbit/s using an unstable channel. We simulate unstable channels by marking an increasing percentage (1% increase per second) of sub-packets as unrecoverable. The channel's current packet loss rate is calculated by the Ack-Processor using the incoming acknowledgments and is analyzed by the sender NIC. The sender NIC compares the current packet loss rate with the configured 10% packet loss threshold. When the packet loss rate exceeds the threshold, the sender NIC uses the protocol template to switch the ongoing transmission to another communication channel. The new protocol description is sent to the receiver which also switches to the new protocol.

The goodput and latency results are shown in figure 2. As expected, the protocol is re-specified whenever the packet loss rate increases over 10%. The reaction time is low enough so that the effective packet loss rate does not increase above 10% (horizontal line in Fig. 2/Goodput) and the goodput and latency are unaffected by the protocol specification and replacement (vertical dashed lines).

IV. RELATED WORK

Adaptable protocols and protocol stacks have been a research field for many years and with the advent of software-defined networking, adaptability became a spotlight research field. However, until now 100 Gbit/s communication was a field for highly specialized protocols and special NICs, while protocol flexibility was no concern. Therefore, we present here selected related work that shares some goals with our approach but are meant for other areas of application.

In [4] the authors use the automatic generation of congestion control algorithms to optimize the performance of TCP multiuser networks. The congestion control algorithms are computed offline with the help of a priori knowledge of the network parameters and the desired output. The computed algorithms outperform human-designed solutions in several cases. The presented approach has the advantage that it does not need changes at the receiver. In contrast, we propose to adapt protocols based on protocol template descriptions, that may change any part of the protocol, i.e., we trade a higher flexibility for the possible necessity to update the receiver. The authors focus on the automatic optimization of the congestion algorithm, while we provide the means to describe protocols as templates. These templates can be specialized for different communication parameters.

The THINK framework [5] is a C based FRACTAL [6] implementation. FRACTAL and THINK allow dynamic binding and seamless replacement of protocol components during run-time. However, components undergoing a reconfiguration must be stopped, which effectively disrupts the processing. Additionally, the authors state in [6] that the reconfiguration process is "not very efficient", which is a key requirement for the reconfiguration in an ultra-high data-rate scenario.

The DRoPS framework [7] provides flexible protocols by combining predefined microprotocols. A protocol's behavior can be changed by replacing a microprotocol with a microprotocol that provides the desired new behavior. Besides establishing a new connection for a changed protocol, DRoPS also provides the possibility to add up to three protocol changes to an outgoing packet. In case an existing connection is reconfigured, the incoming packets must be blocked until the reconfiguration is carried out, which leads to a high overhead.

VirtualStack [8] is an architecture that provides flexible network stacks to applications by using virtual network interfaces to decouple an application from the network stack. By these means, applications can benefit from different network paths without having knowledge of the underlying network. Additionally, VirtualStack provides rule-based programming,

in high data rate communication systems.

that allows defining thresholds, e.g., minimum throughput. When the threshold is reached, an additional channel with a new network stack is created transparently. VirtualStack is meant to add flexibility to general purpose application and does not provide a concept for parallelization. Furthermore, protocols cannot be built from a description.

V. CONCLUSION AND FUTURE WORK

In this paper, we argued the advantages of protocol template implementations that allow the protocol developer to derive different adaptations from a single protocol template by annotating the description with parallelization counts. We introduced the Protocol Engine Template Language and showed how a template unfolds in different template specialization settings.

The evaluation results show that the use of protocol templates is a way to add flexibility to communication systems without adding additional constant overhead. Currently, we are investigating possibilities how resources, such as processor cores, can be assigned on demand in a system aware manner. This will lead to a dynamic resource management that allows us to assign resources automatically to a PETL template description.

ACKNOWLEDGEMENT

This work is part of the End2End100 Project which started in 2013 and is funded by the German Research Foundation (DFG), DFG NO 625/9-1.

REFERENCES

- [1] S. Büchner, L. Lopacinski, J. Nolte, and R. Kraemer, "100 gbit/s end-to-end communication: Designing scalable protocols with soft Real-Time stream processing," in *41st Annual IEEE Conference on Local Computer Networks (LCN 2016)*, Dubai, United Arab Emirates (UAE), Nov. 2016.
- [2] S. Büchner, L. Lopacinski, R. Kraemer, and J. Nolte, "Protocol Processing for 100 Gbit/s and Beyond - A Soft Real-Time Approach in Hardware and Software," *Frequenz*, vol. 71, pp. 427–438, Sep. 2017.
- [3] *TILEncore-Gx72 Intelligent Application Adapter*, Mellanox® Technologies, Product Brief Rev 1.4, 2016.
- [4] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 123–134. [Online]. Available: <http://doi.acm.org/10.1145/2486001.2486020>
- [5] M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polakovic, M. PoulhiÄls, J. Pulou, S. Seyvoz, J. Tous, and T. Watteyne, "Think: View-based support of non-functional properties in embedded systems," in *2009 International Conference on Embedded Software and Systems*, May 2009, pp. 147–156.
- [6] E. Bruneton, T. Coupaye, and J. B. Stefani, "Recursive and dynamic software composition with sharing," 2002.
- [7] R. S. Fish, J. M. Graham, and R. J. Loader, "Drops: kernel support for runtime adaptable protocols," in *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, vol. 2, Aug 1998, pp. 1029–1036 vol.2.
- [8] J. Heuschkel, A. Frömmgen, J. Crowcroft, and M. Mühlhäuser, "Virtual-stack: Adaptive multipath support through protocol stack virtualization," in *Eleventh International Network Conference, INC 2016, Frankfurt, Germany, July 19-21, 2016. Proceedings*, 2016, pp. 73–78.