The Pipeline Concept as Key Ingredient for Modular, Adaptive Communication for Cyber-physical Systems

Stefan Linecker[1], Jia Lei Du[1], Anderson Valões Silva[1] and Reinhard Mayr[2]
[1]*Advanced Networking Center, Salzburg Research, Salzburg, Austria*
[2]*COPA-DATA GmbH, Salzburg, Austria*

Keywords: Cyber-physical Systems, Internet of Things, Interoperability, Self-adaption, Software Architecture.

Abstract: We are currently experiencing another phase within the digital transformation. This phase is prominently called the Internet of Things. It is enabled by the progress in energy efficiency, cost and capability both in sensor-actuator electronics and in data transmission technologies. The envisioned Internet of Things will consist of billions of connected cyber-physical systems. To fully harvest the potential of this development, a strategy for robust, interoperable and future-proof network communication between a myriad of different systems in a global network is required. The ongoing TriCePS project develops both a framework and the missing building blocks to fulfill those requirements. In this paper, the authors propose the pipeline concept as such a building block.

1 INTRODUCTION

A cyber-physical system (CPS) is a system that integrates computation with physical processes. Embedded, networked computers use sensors and actuators to interact with the physical world. Physical processes can then affect computation and vice versa. In contrast to traditional embedded systems, a CPS is a network of interacting appliances (a system of systems if you want) instead of a standalone device (Minerva et al., 2015). The CPS concept is strongly related to another concept called Internet of Things (IoT). IoT describes a global infrastructure of networked everyday objects (or things), which are connected through interoperable communication technologies (Xia et al., 2012). Prominent IoT applications include smart home, wearables, the connected car and many more. CPS and IoT have become possible due to the progress in highly capable, low-cost sensor-actuator electronics and low-cost, energy-efficient, (deeply) embedded computing and communication systems.

The ongoing TriCePS project investigates barriers and possible solutions concerning CPS communication. During our more fine-grained work on the software architecture for the TriCePS project, we identified some core, technical building blocks that were required to make our idea work. This work tries to explain and empathize the importance of pipelines as one of those building blocks.

The remaining part of this chapter gives a brief overview of the TriCePS project, more details can be found in (Du et al., 2019).

1.1 Project Rationale

With a global network of connected entities, there comes a wide range of communication technologies and a myriad of actual connections with varying qualities of connectivity. The most obvious reasons for this volatility being the stochastic nature of packed-switched networks and the greedy nature of applications that compete for available bandwidth. The different qualities of connectivity, a multitude of used communication standards and a wide variety of hardware capabilities create a challenging situation. A generally suitable CPS architecture requires new approaches which can fulfill the adaptivity and interoperability needs for the integration of highly different systems under widely varying conditions.

1.2 Project Overview

The TriCePS project addresses the communication-related challenges in the field of CPS. It aims for adaptive and interoperable communication in such systems. If network conditions vary and the current

Linecker, S., Du, J., Silva, A. and Mayr, R.
The Pipeline Concept as Key Ingredient for Modular, Adaptive Communication for Cyber-physical Systems.
DOI: 10.5220/0009890907750780
In *Proceedings of the 17th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2020)*, pages 775-780
ISBN: 978-989-758-442-8

conditions are not suitable, various strategies can be helpful for sending critical data.

First, some form of prioritization can be used. This typically requires support from and proper configuration of the underlying network and is usually not well supported in networks that are not under your control. Second, it is possible to reduce the data but keep its general type (e.g. through compression or by sending grayscale images instead of color images). Third, one can send a different type of data (e.g. text descriptions instead of a video stream) which can significantly change the amount of sent data.

The software framework developed in the course of the TriCePS project supports all of these approaches. The focus of this paper will be on the second and third approach.

2 RELATED WORK

The overall TriCePS project touches several areas of research in computer networking including network monitoring, network measurements, quality of service and application layer protocols. As stated, CPS communication might require adaptivity and/or modularity. While TriCePS tries to integrate both, related work is mostly found in either one of the categories.

2.1 Adaptivity

The need for adaptivity is not exclusive to the area of CPS. In TriCePS, the application layer is informed about current QoS metrics and has the chance to adapt accordingly. Similar functionality, often called "adaptive codecs" has been around for quite a while. Adaptive Multi-Rate (AMR) audio, which is used in GSM is a classical example. AMR uses different coding schemes depending on link quality measurements (Ekudden et al., 1999). Another prominent and more recent example is a technique called adaptive bitrate streaming (ABS), which is used for video. ABS measures the available bandwidth and adjusts the bitrate (and quality) of the video stream accordingly. Insights into how YouTube is handling this can be found in (Sieber et al., 2016). NADA (Network-Assisted Dynamic Adaptation) is another interesting method since it integrates implicit and explicit congestion signaling (Zhu et al., 2013).

Besides adaptive codecs, switching the communication protocol is another form of adaptivity. This can be important for both interoperability and optimization (using the most appropriate protocol for the current network conditions). A relevant, related project is Application-Layer Protocol Negotiation (ALPN), described in (Friedl et al., 2014). It allows for protocol negotiation within the TLS handshake. The Session Initiation Protocol (SIP), including SDP (Session Description Protocol) can be seen as another example (Rosenberg et al., 2002).

2.2 Modularity

Software architecture-wise, the kind of modularity we were looking for within the TriCePS project could be denoted as building a "communication abstraction layer". The following projects (or parts of projects) or research have similar goals. One relevant implementation is the GNUnet Transport Service API (Grothoff, 2017). "GNUnet is an alternative network stack for building secure, decentralized and privacy-preserving distributed applications." At its very bottom, it uses something called the "transport underlay abstraction" and transport plugins to cope with very different transport mechanisms. Each transport plugin has to provide the means for addressing other entities and sending and receiving of data. Another relevant project is the Common Communication Interface (Atchley et al., 2011), which has its roots in the area of high performance computing. It was designed to provide a common API with support for all major HPC interconnects. Abstraction is provided by the concepts of endpoints, connections, active messages for smaller and remote memory access for larger data movements. The H2020 NEAT project (Grinnemo et al., 2016) is also related. NEAT is a user-level networking API that is agnostic concerning the specific transport protocol underneath. Hiding those details can have benefits concerning the technical evolution of transport protocols. Changes can potentially be made "under the hood" (in the transport protocol), without causing changes in the interface (leaving the user-level application as is). In this respect, NEAT provides very similar abstractions to what Triceps would require to provide a software interface that is future-proof. Another interesting project is the IETF Internet-Draft GEARS, the "Generic and Extensible Architecture for Protocols". It is an architectural proposal that adopts a three layered architecture (hardware drivers layer, generic abstract layer, application protocols layer) and the use of data modelling languages for the development of new protocols (Zhang et al., 2015). Last but not least, we found SensiNact (Gürgen et al., 2016) Bridges to be relevant. SensiNact is a project under the umbrella of the Eclipse Foundcation. It provides an open IoT platform for Smart Cities. SensiNact uses the concept of "bridges" as an abstraction for the interaction with other entities.

Southbound bridges are those that are specialized for the interaction with IoT devices like networked sensors. Northbound bridges are used for managerial tasks and data distribution and implement protocols like MQTT or HTTP/REST.

3 SMART CAMERA REFERENCE SCENARIO

A simple example scenario is presented in Figure 1. A video camera has a network connection to a remote display. The connection will typically cross the public internet. Furthermore, it may use wireless access technologies which will also contribute to variations in the quality of the connection. The camera will use the concepts outlined in the following sections to asses the current network conditions and adapt its strategy of operation. Please note that with this use case, we assume that the video is watched live on the remote display, and therefore, timely playback is more important than consistent quality (which is often the case for monitoring applications). Specifically, the camera will emit high quality, high frame rate video when the network allows it. When network utilization is high, the camera will switch to different modes of operation, e.g. a lower quality, lower frame rate video, single images or even textual representations generated via feature extraction and reconstructed at the receiver.

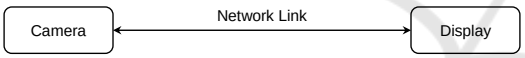

Figure 1: Schematic view of camera use case.

4 PIPELINES FOR MODULAR AND ADAPTIVE COMMUNICATION ARCHITECTURES

The pipeline concept helps making networked applications modular and adaptive. This section describes our thought process concerning modularity before introducing pipelines as a valuable architectural principle.

When designing a networked application, engineers have to carefully consider the network-related abstractions they rely on. The socket API is a well known abstraction in this context but modern software systems frequently use higher-level networking libraries or custom-built networking components. Future-proof networked systems (which are the aim of the TriCePS project) have to be able to adapt to new mechanisms of communication. As a genuine example, consider that a security flaw is found in the communication protocol used by your application. You then need a practical way to update your application. If not considered from the beginning, such changes can be difficult to manage. Therefore, we propose a way of thinking that assumes that all parts of your mechanisms of communication will be replaced some day in the future. We will now try to describe this way of thinking in a more formal manner.

4.1 Problem Formulation

Let's start with the most simplistic model of communication possible. Figure 2 depicts the initial situation. Two communicating applications A and B exchange information (that can be thought of as wire image x[1]).

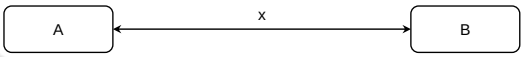

Figure 2: Two communicating applications (A,B) exchanging wire image x.

If we want to change the way this information is represented (the data format or encoding), or change the way this information is encapsulated, protected, acknowledged, etc. (the protocol), we naturally end up with a different wire image on the link between A and B, just as depicted in Figure 3 as y.

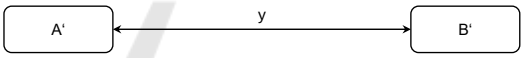

Figure 3: Two modified communicating applications (A',B') exchanging a modified wire image (y).

Depending on the actual change (data format, encoding, protocol, etc.) A and B might have to change too (that is why they are called A' and B' now). The interesting question now is, how to propose generic architectural guidelines so that the difference between A and A' – let's call this difference ΔA has minimal "intertwinedness", or, speaking more in terms of practical software engineering, has well-designed interfaces and abstractions concerning the mechanisms of communication.

Since we highly anticipate changes concerning data formats, encodings or protocols in use (or all or combinations of them), we very much want to have a strategic way to minimize the intertwinedness of ΔA.

[1]The wire image (Trammell and Kuehlewind, 2019) is the sequence of packets sent by each application as seen on a fictive point of observation on the path of communication.

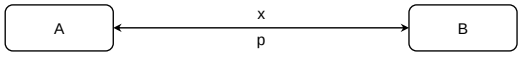

Figure 4: Two communicating applications exchanging wire image x via protocol p.

Let us reconsider our initial situation, where we have A and B that exchange information as wire image x via protocol p (see Figure 4, p is also included there now). We could then add an additional component (we call it a handler, more on that nomenclature later) d, that exchanges wire image x with A via protocol p and wire image y with B via protocol q (just as shown in the Figure 5). The addition of d is transparent to A (it still speaks p and sends/receives x). This is somewhat similar to what protocol converters do (there is no robust definition of nomenclature that is known to us, so the terms protocol converters, protocol bridges and protocol gateways might all be used here. What is meant is a technical artefact, either software or hardware, that can convert between two protocols).

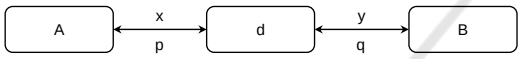

Figure 5: Two communicating applications (A,B) with an additional handler (d) in between.

The protocols p and q might be identical (if only the data format has changed, not the protocol) but x and y will be different in any case (otherwise we would not need this new handler d in between). Now comes the conceptual trick: You can think of d as being embodied in A (in contrary to being an extra component), as shown in Figure 6.

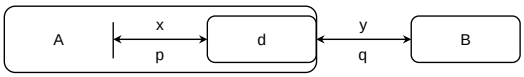

Figure 6: Two communicating applications (A,B), where one (A) has an additional, embodied handler (d).

There are scenarios where the use of an additional handler d is not possible. End-to-end encryption between A and B for example, would make an additional handler d impossible. If, on the other hand, d is embodied in A and end-to-end encryption happens between d and B, you have a design where you can always modify the embodied handler d (we call this modified handler d') in such a way that new requirements from B (e.g. wire image z, protocol r) are met, while maintaining the original interface towards A (wire image x, protocol p), just as illustrated in Figure 7. Additional handlers can (and often will) be stateful and can (and often will) require additional configuration.

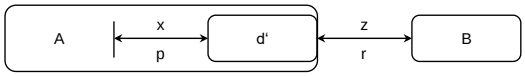

Figure 7: Two communicating applications (A,B), where one (A) has an additional, embodied handler (d') that can communicate with B via a new protocol (r instead of q).

When we design the mechanisms of communication in our software in a way that allows for the addition of auxiliary handlers, we have good chances that future changes can be solved via new or updated handlers. While we discussed individual handlers in the current section, we will now introduce pipelines of handlers.

4.2 Pipelines

The thought model of handlers can help to determine which parts of your application have to be designed in a manner that makes them replaceable but it does not necessarily facilitate the process of doing so. To achieve this, we borrowed the concept of "pipelines" from Facebook's Wangle project[2], which itself adapted this from the Netty (Maurer and Wolfthal, 2016) project. With pipelines "the basic idea is to conzeptualize a networked application as a series of handlers that sit in a pipeline between a socket and the application logic". Figure 8 shows an application where there is no single additional handler d anymore but a whole pipeline (named PL) of handlers (h1, h2, h3, ...).

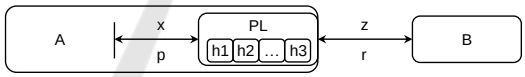

Figure 8: Two applications, one with a pipeline of handlers.

Each handler has a specific duty. For example encryption, framing, compression, character encoding, etc. The aim of this level of modularity is to keep the complexity associated with handling the different communication mechanisms of a wide range of entities manageable. The concrete dissection of the pipeline into individual handlers can of course be a matter of perspective. What we can highly suggest is to follow the now classic UNIX maxim of "do one thing and do it well".

[2]https://github.com/facebook/wangle (accessed January 20, 2020)

5 REFERENCE IMPLEMENTATION

The ideas presented in the former chapters were not born from purely theoretical considerations but from our practical work on a reference implementation of a software library. Figure 10 shows a schematic overview of the library as used by two communicating applications. It can be interpreted with regard to the reference scenario from chapter II. Node A and node B represent two communicating applications. Node A represents the video camera, while node B represents the remote display. Node A and B are connected over the Internet. The video signal captured by A is transmitted to and synchronously displayed by B. Both nodes A and B use of the software library. Furthermore, a new, third entity (entitled repository) comes into play, which acts as a library for new and updated handlers. The business logic of node A feeds video data (denoted as x) into the pipeline. The pipeline uses a set of handlers [3] to process the video data according to current network conditions (as measured by the network monitoring module). The pipeline emits a wire image y (using the protocol p) to the network. Node B receives the data, routes it through all of its pipeline and finally hands over video data (denoted as z) to its very own business logic. Varying network conditions might require the use of a different set of handlers in which case the wire image on the network link (y) will change while the interfaces towards the business logic on both nodes will stay the same.

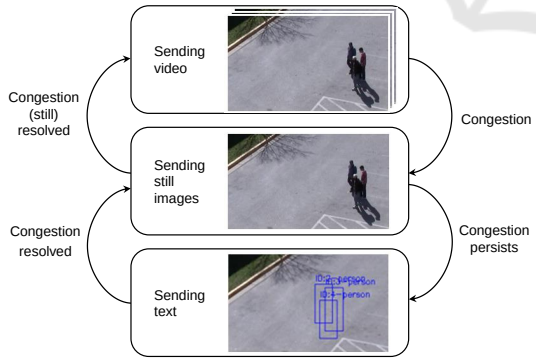

Figure 9: State machine for an adaptive camera.

Figure 9 shows this mechanism as a state machine. When network conditions are good, high quality video[4] will be emitted. When congestion is detected, lower quality still images will be emitted and when conditions get even worse, a text representation of the moving parts of the image will be emitted. With bettering conditions, the same process will happen in reverse order. The state where video is emitted uses a pipeline with two handlers (transcode video to target bandwidth, apply framing), the state where still images are emitted uses a pipeline with three handlers (downsampling, apply image compression, apply framing) and the state where text is emitted uses a pipeline with four handlers (use feature extraction, filter out irrelevant features, apply run-length encoding, apply framing). While video emission will require 2000 kbit/s of bandwidth, text will only require 20 kbit/s. This means, that for this example, we can scale network usage by factor 100, while still providing relevant information on the display on the receiver side.

Apart from the concrete example, the library functionality can be described as three modules. The Network Monitoring module continuously monitors the network flows between A and B and supplies network metrics to both the library and the business logic. The Pipeline module glues together the individual handlers. Data flows from the business logic of a node through all handlers in the pipeline, then through the network to the opposite node, where all handlers are traversed in reverse order. Finally, the Protocol Negotiation module makes sure that the applications use pipeline setups that are compatible with each other while also taking care of the process of retrieving new or updated handlers from the repository.

6 CONCLUSIONS

A pipeline-based design of an application layer communication channel can help with making the mechanisms of communication much more adaptable. We used this approach to develop a framework, software library and demo application that can switch seamlessly between handlers. In contrast to other approaches, TriCePS integrates short-term (think seconds) and long-term (think years) adaptability and modularity into one. Pipelines proved to be a valid conceptual enabler for sound and practical dissection of networked software into lightweight, exchangeable components (handlers).

[3] It is noteworthy that Figure 10 shows one specific, simplified example of a pipeline setup. The two pipelines in the example hold the same handlers (h1, h2, h3, h4) for each node. That is not a requirement. Quite contrary, the two pipelines could each hold a different set and number of handlers.

[4] as provided by (Oh et al., 2011)

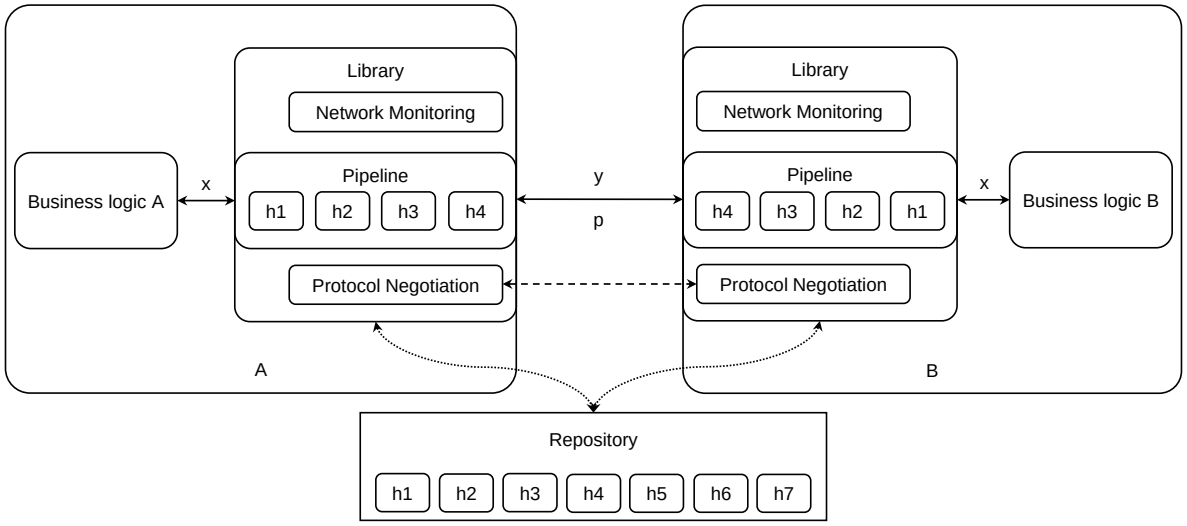

Figure 10: Schematic overview of the reference implementation and its use by two applications.

ACKNOWLEDGEMENTS

This project is partially funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" (https://iktderzukunft.at/en/).

REFERENCES

Atchley, S., Dillow, D., Shipman, G., Geoffray, P., Squyres, J. M., Bosilca, G., and Minnich, R. (2011). The common communication interface (cci). In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, pages 51–60. IEEE.

Du, J. L., Linecker, S., Dorfinger, P., and Mayr, R. (2019). Triceps: Self-optimizing communication for cyber-physical systems. In *Proceedings of the 4th International Conference on Internet of Things, Big Data and Security (IoTBDS 2019)*, volume 1, pages 241–247.

Ekudden, E., Hagen, R., Johansson, I., and Svedberg, J. (1999). The adaptive multi-rate speech coder. In *Speech Coding Proceedings, 1999 IEEE Workshop on*, pages 117–119. IEEE.

Friedl, S., Popov, A., Langley, A., and Stephan, E. (2014). Transport layer security (tls) application-layer protocol negotiation extension. RFC 7301, RFC Editor.

Grinnemo, K.-J., Brunstrom, A., Jones, T. H., Fairhurst, G., Hurtig, P., and Ros, D. (2016). Neat-a new, evolutive api and transport-layer architecture for the internet.

Grothoff, C. (2017). *The gnunet system*. PhD thesis.

Gürgen, L., Munilla, C., Druilhe, R., Gandrille, E., and Botelho do Nascimento, J. (2016). sensinact iot platform as a service. *Enablers for Smart Cities*, pages 127–147.

Maurer, N. and Wolfthal, M. (2016). *Netty in Action*. Manning Publications New York.

Minerva, R., Biru, A., and Rotondi, D. (2015). Towards a definition of the internet of things (iot). *IEEE Internet Initiative*, 1:1–86.

Oh, S., Hoogs, A., Perera, A., Cuntoor, N., Chen, C.-C., Lee, J. T., Mukherjee, S., Aggarwal, J., Lee, H., Davis, L., et al. (2011). A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160. IEEE.

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. (2002). Session initiation protocol. RFC 3261, RFC Editor.

Sieber, C., Heegaard, P., Hoßfeld, T., and Kellerer, W. (2016). Sacrificing efficiency for quality of experience: Youtube's redundant traffic behavior. In *IFIP Networking 2016*.

Trammell, B. and Kuehlewind, M. (2019). The Wire Image of a Network Protocol. RFC 8546, RFC Editor.

Xia, F., Yang, L. T., Wang, L., and Vinel, A. (2012). Internet of things. *International Journal of Communication Systems*, 25(9):1101–1102.

Zhang, M., Dong, J., and Chen, M. (2015). Gears: Generic and extensible architecture for protocols. Draft IETF.

Zhu, X., Pan, R., Ramalho, M., Mena, S., Jones, P., Fu, J., and D'Aronco, S. (2013). Nada: A unified congestion control scheme for real-time media. Draft IETF.