Deliverable D2.1

Piccolo Node definition

WP Leader:	Chris Adeniyi-Jones – Arm Ltd
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public
Contractual delivery date:	March 31, 2021
Actual delivery date:	March 30, 2021
Suggested readers:	Researchers, developers and technologists interested in edge computing and the convergence of networking and computing.
Version:	1
Total number of pages:	48
Keywords:	In-network computing, node

Abstract

This document sets out the high-level design of a Piccolo Node. We describe the design goals, the technologies we envisage using for implementation, some of the existing hardware/software platforms that will be used as Piccolo Nodes and a high-level overview of how the node will operate.

Disclaimer

This work was done within the EU CELTIC-NEXT project PICCOLO (Contract No. C2019/2-2). The project is supported in part by the German Federal Ministry of Economic Affairs and Energy (BMWi) and managed by the project agency of the German Aerospace Center (DLR) (under Contract No. 01MT20005A). The project also receives funding awarded by UK Research and Innovation through the Industrial Strategy Challenge Fund. The project is also funded by each Partner.

Impressum

[Full project title] Piccolo: In-network compute for 5G services
[Short project title] Piccolo
[Number and title of work-package] WP2. Piccolo Node
[Number and title of task] T2.1 Piccolo Node Definition
[Document title] D2.1 Piccolo Platform design specification
[Editor: Name, company]
[Work-package leader: Name, company] Chris Adeniyi-Jones, Arm

Copyright notice

Executive Summary

Work-package 2 will specify, produce and evaluate the Piccolo Node - that is, the capability of an individual node in a Piccolo network. The Piccolo Node Platform will deliver an open, low latency, efficient, secure, in-network compute implementation.

In this deliverable D2.1 "Piccolo Platform Design Specification" we focus on the essential features of a Piccolo node and describe the design of the framework of APIs and protocols required to implement a Piccolo Node.

List of Authors

Organisation	**Author**
Arm Ltd.	Chris Adeniyi-Jones
British Telecommunications plc	Philip Eardley, Andy Reid, Peter Willis
Fluentic Networks Ltd.	Ioannis Psaras, Alex Tsakrilis
InnoRoute GmbH	Andreas Foglar, Marian Ulbricht
Robert Bosch GmbH	Dennis Grewe, Naresh Nayak
Sensing Feeling	Dan Browning, Jag Minhas, Chris Stevens
Stritzinger GmbH	Mirjam Friesen, Sascha Kattelmann, Peer Stritzinger, Stefan Timm
Technical University Munich	Vittorio Cozzolino, Raphael Hetzel, Nitinder Mohan, Jörg Ott
University of Applied Science Emden/Leer	Dirk Kutscher, Laura al Wardani

Table of Contents

List of Figures

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BPF	Berkeley Packet Filter
CPU	Central Processing Unit
DSRC	Dedicated Short Range Communication
EE	Execution Environment
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
IP	Internet Protocol
IPv6	IP Version 6
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
MAC	Media Access Control
NUMA	Non-uniform Memory Access
OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
OTP	Open Telecom Platform
P4	Programming Protocol-Independent Packet Processors
PCIe	Peripheral Component Interconnect Express
PDP	Programmable Data Plane
SDN	Software Defined Network
SPI	Serial Peripheral Interface
SSD	Solid-State Drive
TEE	Trusted Execution Environment
URL	Uniform Resource Locator
VM	Virtual Machine
YANG	Yet Another Next Generation

Definitions

TrustNode: Hardware accelerated Routing devices that include Field Programmable Gate Array (FPGA) for fast routing and supporting Central Processing Unit (CPU) for advanced packet processing. More information see: `http://TrustNo.de`

1 Introduction

This document sets out the high-level design of a **Piccolo Node**. We describe the design goals, the technologies we envisage using for implementation, some of the existing hardware/software platforms that will be used as Piccolo Nodes and a high-level overview of how the node will operate. In this context a Piccolo Node is a computing system controlled and managed via Piccolo Application Programming Interfaces (APIs) to provide distributed compute/network functionality. A **Piccolo System** consists of Piccolo Nodes that are connected and can communicate with each other.

Deliverable D1.1 "Use cases, Application Designs and Technical Requirements" [1] summarises the use cases for in-network computing and provides resulting requirements for WP2 and WP3. In deliverable D3.1 "Architectural Invariants for Distributed Computing" [2] we document the abstractions and the invariants that will form the basis for Piccolo's distributed computing system which a collection of Piccolo nodes will provide. In this deliverable D2.1 "Piccolo Platform Design Specification" we focus on the essential features of a Piccolo node.

There are many options to be considered when building a Piccolo system and nodes. It is our intention to be technology/protocol agnostic where possible thus allowing a wide-variety of Piccolo nodes to be built from hardware/software platforms ranging from small embedded systems to cloud-compute capable server machines. The exact nature of the network connecting Piccolo nodes to each other is not specified in this document.

2 State of the art/General Technology

In this section we describe key software technologies which we believe could be used in the implementation of a Piccolo Node. This is not an exhaustive list but does reflect the expertise of the project partners. It is also not prescriptive - a Piccolo node may include only some or even none of these components.

2.1 Containers

OS-level virtualization is an operating system paradigm in which the kernel allows the existence of multiple isolated userspace instances. These run and rely upon the underlying host system meaning that every single one also shares the same host kernel through the virtualization engine. Instead of virtualizing all the physical hardware, only the software stack sitting on the kernel is virtualized. As a consequence, OS-level virtualization offers little to no overhead as the containers are running directly on top of a shared kernel. This form of virtualization is often called *containers* of which some popular implementations are Docker [3] and LXC [4].

Linux containers are commonly used to provide a portable, consistent environment for running applications. The view of the system from the applications running inside a container is modified by the use of Linux control groups (cgroups) and Linux Namespaces, features provided by the Linux kernel. The environment inside the container appears to be an independent instance of the operating system. Processes running inside a container are isolated from processes running in other containers and processes running on the host system. Resources such as CPU, memory and access to devices, available to each container can be controlled. Containers are very lightweight when compared to virtualization solutions using System virtual machines because using containers requires only one operating system kernel. From a security perspective the sharing of a single operating system kernel is potential security vulnerability when using Linux containers.

There are now solutions that attempt to bridge the gap between full-system virtualization and cgroup/-namespaces based containerization. kata-containers is "an open-source runtime building lightweight virtual machines that seamlessly plug into the containers ecosystem." [5]. kata-containers use hardware-virtualization to run a dedicated kernel for each container providing isolation of network, I/O and memory and can utilize hardware-enforced isolation. The kernel that is executed for each container is not the same as the host-kernel and a simpler root-filesystem is provided.

A more detailed discussion of containers can be found in [6], a preliminary version of which served as input to this section.

2.2 Unikernels

Radical OS architectures from the 1990s introduced the concept of *library operating system* (libOS). In a libOS, protection boundaries are pushed towards the lowest hardware layers, providing: (*i*) an ensemble of libraries to interact with hardware or network protocols, and (*ii*) rulesets to set protection boundaries for the application layer. Some examples are Exokernel [7] and Nemesis [8]. A few advantages of libOS are increased performance due to reduced context switch between user and kernel space, tiny attack surface compared with VMs and containers, fast boot-up time, and extremely small footprint with only around 4% the size of the equivalent code bases using a traditional OS [9]. However, libOS also have drawbacks among which the biggest ones are additional complexity in isolating multiple applications resources and the cost of rewriting device drivers to fit the new architecture. The advent of hypervisors helped in mitigating this issues and opened the doors to unikernels. In particular, para-virtualization brought to unikernels virtualized disk and network drivers, interrupts and timers, emulated motherboard and legacy boot, and privileged instructions and page tables. Having a single address space yields improved performance and a simpler software design since repeated privilege transitions (and possibly copying of data) between user space and kernel space are not necessary anymore. However, without address space separation running multiple applications side by side with additional resource isolation can become complex.

Unikernels are single-purpose appliances that are at compile time specialized into standalone kernels [10], and sealed against modification after deployment. They are written in a high-level language and act as individual software components. Generally, a full application consists of a set of running unikernels working together as a distributed system [11].

They can be seen as an extreme form of lightweight virtualization emerged from the observation that most virtualized applications are often burdened by additional, unnecessary functionalities and libraries inherited from the underlying OS. For example, VMs have many software layers while, ultimately, performing a single function such as database or Web service. This represents a real opportunity for optimization both in terms of performance, by adapting the virtual instance to its task, but also for improving security by eliminating needless functionality. Their attack surface is strictly confined to the running application logic as everything that is part of the unikernel is directly compiled into the application layer. Therefore, each unikernel may have a different set of vulnerabilities, implying that an exploit that can penetrate one may not be threatening the others. However, the high degree of specialization means that unikernels are unsuitable for general purpose applications. Adding functionality or editing a compiled unikernel is generally not possible, and instead the approach is to compile and deploy a new unikernel with the desired changes.

Initially designed for public clouds, unikernels are also potential virtualization candidates for edge-cloud networks due to their small footprint and flexibility, as shown in other research efforts [12]. Unikernels have been primarily designed to be stateless, similarly to lambda functions. Therefore, they are a good fit for standard stateless functional algorithms [13] or for Network Function Virtualization (NFV) [14]. There are multiple available unikernel implementations which differ mainly in the supported programming language. MirageOS [10] is a unikernel based on the OCaml functional language which aims at unifying both kernel and application userspace into a single, high-level framework. Among other benefits this brings static type-checking, automatic memory management, modularity and meta-programming (optimizing compiled code based on runtime parameters). It has been used in multiple research works exploring the tradeoff of using unikernels for edge computing [15, 16, 17]. HaLVM [18] is an unikernel based on Haskel with pervasive type-safety in the running code. IncludeOS [19] and ClickOS [20] support C++ with the former coming in the form of a framework to which is possible to bind any application. The former has been used to deploy lightweight security solutions at the edge showing excellent performance compared to well-known tools [21]. The latter is highly specialized in offering functions to do network traffic processing (based on the Click modular router [22]). OSv [23] is a unikernel offering more flexibility supporting multiple runtimes and languages such as the Java Virtual Machine (JVM). The GRiSP platform [24] provides a toolchain to generate unikernels based on the Open Telecom Platform (OTP, see 3.3) and RTEMS [25], a real-time library OS.

Ultimately, unikernels are good candidates for the creation of systems based on microservices and serverless architectures. This is especially true at the edge, where we can benefit the most from the unikernels perks such as small memory footprint and reduced attack surface. Unikernels have a special relation to the world of IoT which puts rather harsh requirements on appliances. Software services are supposed to be very resource efficient with a small memory footprint while at the same time having (soft) real-time processing capabilities, and they should be able to spawn and die quickly such that orchestration mechanics work instantaneously. All these properties are typically present in unikernels.

A more detailed discussion of containers can be found in [6].

2.3 Programmable Data Planes

With the success of the software-defined networking paradigm, the networking community is striving to further increase the programmability of networks. The introduction of Programmable Data Plane (PDP) such as Programming Protocol-Independent Packet Processors (P4) [26] aims to overcome the last limitations (e.g., static configuration of packet processing) by realising fully programmable networks. PDPs enable programming the behaviour of network elements such as programming packet processing pipelines in the data plane devices, while also permitting runtime configuration of these devices as in Software Defined Network (SDN). Some of the key aspects of a programmable data plane design include:

- **Non-Rigidity**: The underlying hardware Application-Specific Integrated Circuit (ASIC) is no longer rigid with a programmable data plane. The user can program it dynamically.
- **Abstraction & Interoperability**: The program used to configure the ASIC has to be portable in the sense that with little or no modifications it can run on a hardware from a different vendor. Thus, a higher level of abstraction can be achieved.

- **Modularity**: The hardware elements should be available as building blocks, which can be used by the programmer according to the needs.

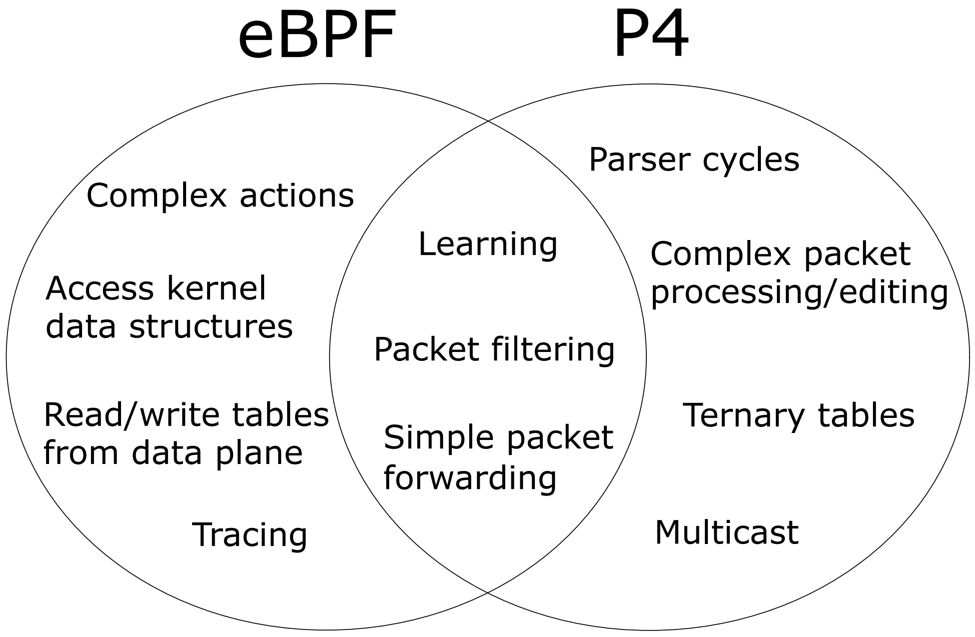

Figure 1: Due to the different expressiveness of the eBPF and P4 programming languages, only a subset of features can be realised in eBPF and cross-compiled to P4 targets. Contents of the Figure are based on [27].

Programmable data planes are implemented in both software and hardware solutions. Example for a software solution is Click modular router [28]. It consists of several modules that can be treated as building blocks and the user could pick them and build custom packet processing pipelines. A variety of increasingly complex functions can be implemented using this technique. Detailed information about hardware solutions of programmable data planes are provided in Piccolo Deliverable 3.1 [2].

From a Piccolo node perspective, software based data plane solutions and libraries can be incorporated within the node architecture supporting mechanisms to allow a Piccolo node to influence network behaviour such as packet forwarding and processing. Instead of relying on specific hardware, *packet processing libraries* allow for network process acceleration by bypassing the kernel of the underlying operating system. The Data Plane Development Kit [29] is a set of libraries that allows to improve the processing performance of network traffic on a variety of CPU architectures.

Another solution executed in the Linux kernel space is RedHat's eXpress Data Path (XDP) framework that allows to implement custom extended Berkeley Packet Filter (BPF) [30] programs attached to the kernel processing in the stack. BPF introduces a virtual machine within the Linux operating system kernel for which custom programs can be developed and executed within the OS kernel for faster processing. However, due to the different expressiveness of the BPF and the P4 programming languages, only a subset of features can be realised in extended BPF and cross-compiled to P4 targets (cf. Figure 1).

Another option to implement packet processing is the P4 technology. P4 can be seen as a modelling language to express the forwarding function in a packet processing pipeline. A P4 program can be compiled down to various platforms ranging from software switches (e.g., into BPF programs) to re-configurable architectures like FPGAs. It also provides constructs to express minimal computing tasks in the forwarding functions enabling offloading of computations into the network [31]. This possibility to push computations from applications to the network with no (or minimal) effect on latency or throughput can go a long way in improving the overall system performance.

2.3.1 Computing applications using P4

Many of the application layer frameworks are based on generic functionalities (like distributed consensus) implemented high up in the networking stack. This affects the overall throughput of the system because of the bottleneck in the application layer. Additionally, developers have to deal with implementing these functionalities and integrating it with the application logic. PDPs enables bringing many of these applications lower into the networking stack. NetPaxos [32] and NetCache [33] implement consensus algorithms and distributed key values stores respectively, both critical for implementing distributed systems. There have been several noteworthy attempts to implement complete applications into the network. Rüth et al. demonstrate execution of a feedback control loop using extended BPF described using P4 language [34]. Edge detection, serving as a primitive for complex image processing applications, has been also implemented using P4 [35]. P4CEP presents a complex event processing framework implemented in the networks using smart network interface card [36].

2.4 Trusted Execution Environments

Trusted Computing relies on a separate (external) hardware module that offers a functional interface for platform security. This Trusted Platform Module (TPM) [37] allows a system to prove its integrity and to conserve cryptographic keys inside a tamper-evident hardware module. The main flaw of the TPM is that it does not offer an isolated environment for different (external) entities, thus restricting its functionality to a limited set of APIs.

The most notable effort to address this issue is to enable the execution of arbitrary code in a circumscribed environment, which provides tamper-resistant execution to the applications. The isolated environment is called a Trusted Execution Environment (TEE) and is defined as a secure, integrity-protected processing environment, consisting of memory and storage capabilities [38].

A core aspect of the TEE is the separation kernel that ensures isolated execution. Its fundamental design purpose is to enable the coexistence of various systems demanding different levels of security on the same platform. In other words, it splits the system on several partitions and ensures that all are segregated from each other, except for an inter-partition communication interface that is carefully designed to allow the exchange of messages. Unlike traditional kernels such as Operating Systems, hypervisors and micro-kernels, the separation kernel is not complicated and provides both time and space partitioning, making it the most vital part of the TEE. A set of requirements is described in the

Separation Kernel Protection Profile (SKPP) [39]:

Data separation: Data inside a partition are not allowed to be read or modified by other partitions.

Sanitization: Resources that are shared between the partitions cannot be used to leak data.

Information flow administration: Explicit permission is required for partitions to communicate with each other.

Faults isolation: If a security breach occurs in one partition, it should not propagate to other partitions.

In principle, Trusted Execution Environments guarantee:

- Authenticity of the executed code
- Integrity of the runtime states (CPU registers and delicate I/O operations among others)
- Confidentiality of the data, code and runtime states stored on a persistent memory.

The above guarantees are extended by features such as remote attestation which provides trustworthiness of the (sealed) executed code for third parties. In general, the threat model of a TEE incorporates all software and physical attacks performed on the main memory and its non-volatile memory by a privileged adversary (from the OS to the administrator of the system).

2.4.1 Technologies

- Intel Software Guard eXtensions (SGX)

  Intel (SGX) [40] is an example of TEE technology that enables applications or part of the application to be executed in an isolated environment. SGX is an architectural feature that provides a new set of CPU instructions that allow a user application to generate and use the hardware-assisted TEE referred to as an enclave.

  The enclave code and data are located in a domain of protected physical memory that is defined as the Enclave Page Cache (EPC). The EPC size can be set between 32MB, 64MB or 128MB [41]. Code and data that reside in cache are protected by CPU access controls. When code and data that reside in the EPC pages are loaded into DRAM, then they are shielded at a cache line level. Cache lines in the EPC are encrypted and decrypted to and from the DRAM using an on-chip module named Memory Encryption Engine (MEE). Modifications or rollbacks in the enclave memory are detected due to its inherent integrity protection mechanisms.

  A special property is that non-enclave code cannot access enclave memory, while enclave code has the liberty to access untrusted DRAM outside the EPC directly. This is done in order to deliver function call parameters and responses. However, the integrity of the untrusted data (function call results) needs to be verified by the enclave code.

Another feature of the Intel SGX is remote attestation. Remote Attestation evaluates the enclave identity, the integrity of the code inside an enclave, its structure, and ensures that a genuine Intel SGX processor is executing the enclave. Additionally, the procedure provides the initial shared secret between the service provider and the enclave application to facilitate the construction of a trusted communication channel via untrustred networks.

- AMD Secure Encrypted Virtualization (SEV)

  The AMD SEV is a security feature of the AMD Memory Encryption Technology and was primarily designed for the public cloud where cross-VM and hypervisor-based attacks are a critical issue. It mainly addresses attacks stemming from the high-privileged system softwares, by providing encrypted Virtual Machine (VM) isolation. AMD SEV encrypts and shields the VM's memory space with the VM's distinct encryption key from the hypervisor or other VMs on the same platform. Therefore, the primary responsibilities of SEV are runtime protection and secure initialization of VMs. The core of SEV's runtime protection is a memory encryption engine located in the memory controller that encrypts the main memory using AES-128 [42]. In addition, SEV protection is transparent to user application software, making it a suitable TEE for shielding legacy software applications. Although SEV supports unmodified enterprise-level applications, it includes the underlying OS and hypervisor in the Trusted Computing Base (TCB), thus making it vulnerable to a large class of attacks. Therefore, SEV is not suitable for applications that require a significant-degree of security protections.

- Arm TrustZone

  The Arm TrustZone hardware architecture [43] aims to provide a security framework that enables a device to counter many of the specific threats that it will experience. The primary security objective of the architecture is to enable the construction of a programmable environment that allows the confidentiality and integrity of almost any asset to be protected from specific attacks. The security of the system is achieved by partitioning all of the SoC's (System on Chip's) hardware and software resources so that they exist in one of two worlds - the Secure world for the security subsystem, and the Normal world for everything else. Hardware logic present in the TrustZone-enabled bus fabric ensures that no Secure world resources can be accessed by the Normal world components, enabling a strong security perimeter to be built between the two. The second aspect of the TrustZone hardware architecture is the extensions that have been implemented in some of the ARM processor cores. These additions enable a single physical processor core to safely and efficiently execute code from both the Normal world and the Secure world in a time-sliced fashion. In this way a TEE can be run in the Secure world, with its code and data isolated from the Normal world. TrustZone can be used to implement a Secure Boot process with all boot code verified via public and private keys providing a chain of trust. The TEE running in the Secure world can then provide services such as key management or Digital Rights Management, to "untrusted" software.

2.5 Language Virtual Machines

Compared to system virtual machines that execute both the operating system and applications, process virtual machines execute a single application [44]. While there is a class of these process virtual machines that executes real Instruction Set Architectures (ISAs) and Application Binary Interfaces (ABIs), the class of high-level language virtual machines is focused on executing portable and system-independent instruction sets and APIs [44]. Systems combining these high-level language virtual machines with features such as garbage collection are also referred to as Managed Runtime Environments [45]. Examples for this class of systems and instruction sets are the Java Virtual Machine (JVM) with its bytecode format [46] and the recent WebAssembly [47] format.

The virtual machines execute these applications using interpretation or translation to the target ISA, and provide them with an API to interact with the host system [44]. As they are able to tightly control these system interfaces and the code's execution, these language virtual machines may act as an isolation boundary for untrusted applications. Furthermore, some language virtual machines such as V8 [48] provide additional sandboxing functionality that allows them to run multiple applications in the same virtual machine in an isolated way while also amortizing its resource overheads. Therefore, they are used to isolate, e.g., websites executing code in the browser [47] or multiple serverless applications [49].

2.6 Erlang VM

The Erlang VM represents the core utility of the Erlang Open Telecom Platform [50] and can be described as a language virtual machine that executes programs based on a guest language. The Erlang VM itself provides its own virtual instruction set architecture with higher level semantics. The guest language is presented to the Erlang VM in form of bytecode which is executed within a register machine (also called BEAM - Bogdan/Björn's Erlang Abstract Machine) where all instructions operate on named registers. Typically the guest language is the Erlang language itself but this abstraction made the creation of various other languages (Elixir, LFE, IEC 61499 etc.) possible. Most of these languages rely on one of the well defined layers (Erlang abstract format, Core Erlang or Erlang static single assignment format) that serve as an intermediate representation of program code suited for further compilation into executable bytecode.

Erlang program code is executed within Erlang processes which are comparable to green threads known from other programming languages. An Erlang process is technically a low memory footprint stack of frames that is handled in user space by the Erlang VM allowing very efficient context switching. There is no memory sharing between Erlang Processes.

The Erlang VM is known for its outstanding soft real-time aptitude which is largely enabled by its scheduling capabilities. It also supports symmetric multiprocessing (SMP) in the sense that all available CPU cores can be used. Typically there is one Erlang scheduler running within an OS thread for each CPU core and all schedulers are managed by a centralized facility which distributes workload in

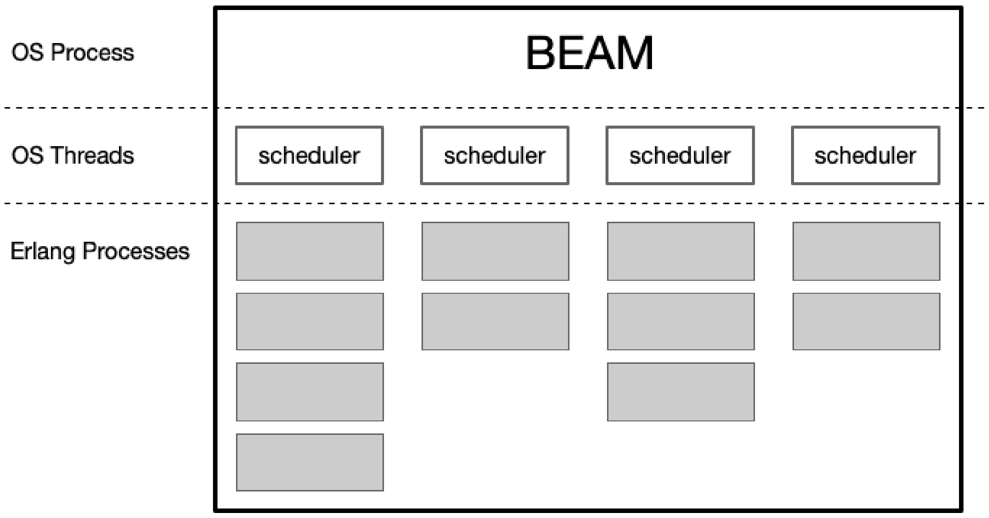

Figure 2: Erlang VM Architecture

a round robin fashion onto a work queue per scheduler.

Erlang schedulers are preemptive schedulers, e.g. the scheduler is capable of pausing a process after a certain amount of configurable work is done such that another process from the work queue of the scheduler can be executed. This amount of work is measured in 'reductions' and the scheduler can pause a process after such a reduction. A reduction is a very fine-grained unit of work and can be e.g. a function call or I/O operation. This combination of preemptive scheduling and SMP enabled fine-grained workload management is to date still unparalleled by other comparable language VMs or integrated scheduling mechanisms (the closest being the golang scheduler).

The Erlang VM is portable and able to run on various OS (Linux, BSD, Windows, etc.) and remarkably even 'bare metal' on microcontrollers [24].

2.7 OPC Unified Architecture

2.7.1 General description

OPC Unified Architecture (OPC UA) is an general, platform independent data infrastructure solution. Mainly driven from industrial context, OPC UA provides an general framework for exchanging management and process data between devices. Figure 10 gives an overview about architecture and provided interfaces. OPC UA provides two ways of data access: the classical request response approach and an event driven publish subscribe interface. The base of the OPC UA communication is the information model which describes the data provided by the OPC UA servers via the different interfaces. The tree structured information model is typical defined in some database language e.g. Yet Another Next Generation (YANG). The OPC UA server provides his application data according to this information model. Client can request parts or the whole model. Alternatively objects of the application data can be subscribed according the publish subscription interface. The communication

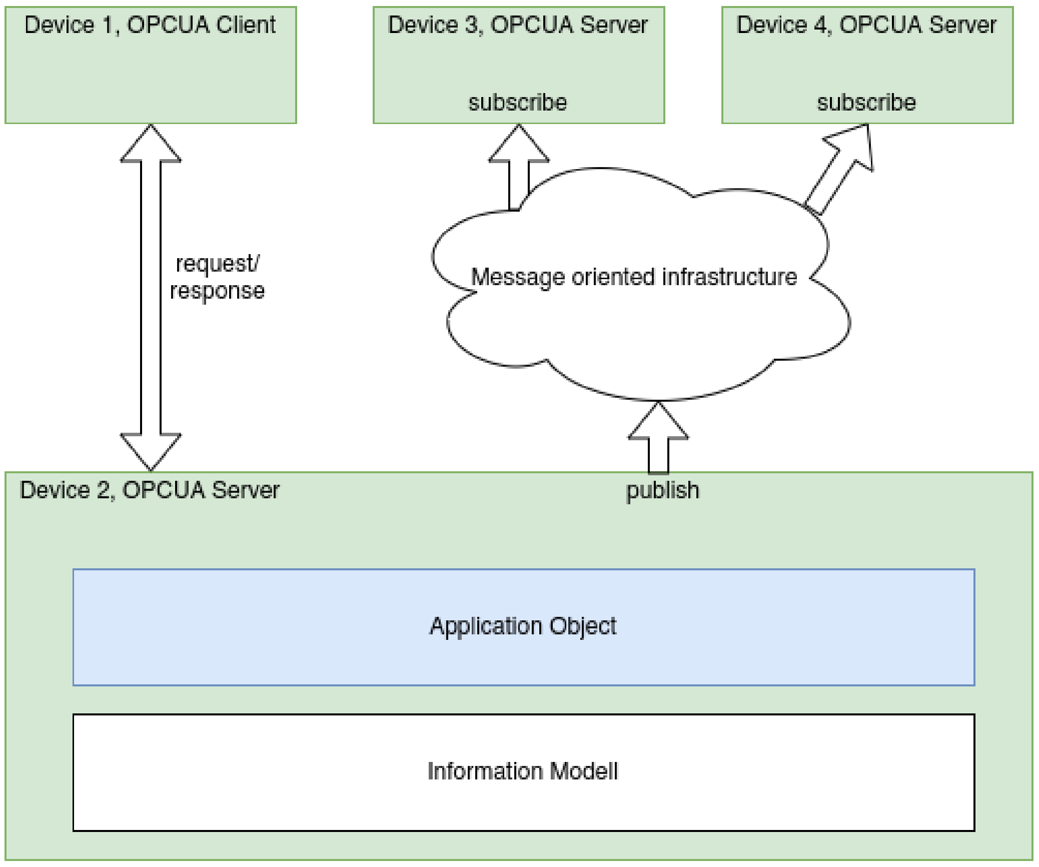

Figure 3: OPC UA general architecture [51]

itself can be text/http based or binary according to the resources of the used infrastructure.[52]

2.7.2 OPC UA in Piccolo context

There are several advantages why the use of OPC UA as middleware within the framework of Piccolo can be considered. In particular, the Comprehensive information modeling functionality via tree structured data models of OPC UA has sufficient semantic coverage to be used as a mediator of the different technologies. Besides the information model, OPC UA offers two different discovery services to find and explore other OPC UA instances in the network via a discorvery server. At the same time, such a discovery server can be used for the presentation of available technologies and resources. So OPC UA should be ideally suited for the exchange of control data and metadata.

3 Different platforms in Piccolo

The Piccolo project envisages that the role of a Piccolo node could be fulfilled by many different hardware/software platforms. We expect that it will be possible to build a Piccolo system in which the nodes are heterogeneous, varying in form-factor and capability. In this section we describe the platforms (hardware and software) being used by the partners within the project that we consider to be representative of the variety of platforms that could function as Piccolo nodes. We expect our initial proof-of-concept implementations to be based on one or more of these platforms.

3.1 Sensing Feeling Containers

The visual processing use case relies on data being processed at many different locations. At the edge, there is a camera recording footage. The images produced are then sent over the gRPC protocol [53] – optimised protocol for remote procedure calls – to the Visual Processing Engine (VPE) in the Fog for image inference, and the results of this are sent onto the Cloud for further analysis. Distributing the process like this requires a flexible, dynamic solution in the form of containers.

The container at the edge should be as simple as possible. In effect, its only job is to forward images onto the Piccolo Node. The edge device hosting the container can be a low power, low cost chip, which gives the system flexibility to be used in all kinds of scenarios and enables use cases where the device could even be considered as expendable. The images produced from the device will be securely transferred into the container running inside a Piccolo node, where initial algorithms will be processed. These algorithms are dependent on the configuration of the transmitting device, based on the use case - for certain scenarios this could be person detection, vehicle detection, emotion and head pose, etc. The majority of these algorithms will be stateless, transforming a single image into a single text output. This data is then forwarded on again, to the final containers in the cloud, where this raw data is stored. Here, the data undergoes another round of analysis, carrying out more complicated, stateful computations such as detecting motion paths and raising system alerts based on aggregated data.

Utilising containers allows the system to have minimal downtime across the board, and gives full control of when and where specific code is deployed. Alongside these benefits, containers are mostly hardware and infrastructure agnostic - no matter what device is running the code, the functionality and output should remain consistent. This consistency is key when working with edge devices; the software should execute correctly no matter the specification of the device. Containerisation also makes it incredibly easy to update any individual part of the system at any time. If an update is required for analytics in the Cloud, there is no reason this should cause downtime on the VPE, for example. Containers are designed to run as isolated processes, meaning that multiple containers can be run on a single machine without interfering with each other. This means a single node could hypothetically run multiple VPEs, with each servicing different edge devices with different algorithms, and each VPE could be completely ignorant of the others.

With communication being such an integral part of this process, it's important that data is transmitted securely between each of these containers. Encryption and authorisation should be a top priority. Data should be encrypted at rest and during transmission, any unauthorised parties should have no access to the inner workings of the containers, and identities of edge devices must be preserved.

3.2 Apache Flink

Apache Flink is a framework designed for processing stateful computations over both bounded and unbounded data. It is designed to be massively scalable, with high performance, and able to run in all common cluster environments. It has been proven to be capable of processing trillions of events per day, maintaining terabytes of state and running on thousands of cores. Its ability to process unbounded data makes it perfect for working with continuous input and data streams, specifically in the cases of event-driven applications, data analytics and data pipelines.

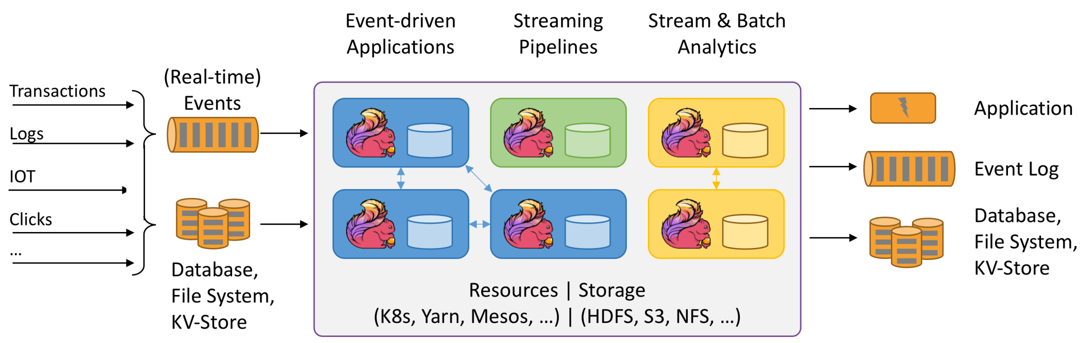
Event-driven Applications
Streaming Pipelines
Stream & Batch Analytics
Transactions
Logs
IOT
Clicks
...
(Real-time) Events
Database, File System, KV-Store
Resources | Storage
(K8s, Yarn, Mesos, ...) | (HDFS, S3, NFS, ...)
Application
Event Log
Database, File System, KV-Store

Figure 4: Different Flink Applications

A key functionality of Flink is its automated checkpointing, providing rollbacks after failures and guaranteeing exactly-once state consistency. These periodic backups allow Flink to seamlessly recover from faults, restoring state and identifying the stream position to continue processing from in the case of failure. This guarantees an exactly-once approach to the application state, resulting in consistent application state even when hitting multiple failures. However, this is not the same as exactly-once processing, so care must be taken to avoid reprocessing large amounts of data and potentially sending duplicate events out the other side.

3.3 Erlang/OTP

The Open Telecom Platform (OTP) [50] is aimed at providing efficient development and operation of robust, adaptable distributed systems based on the Erlang programming language. It consists of an Erlang runtime system, a number of ready-to-use components mainly written in Erlang, and a set of design principles for Erlang programs. Since Erlang and OTP are closely interconnected the term Erlang/OTP is normally used instead of OTP. Erlang is a general-purpose programming language with built-in support for concurrency, distribution and fault tolerance. Erlang's process model resembles the actor model and uses lightweight user space processes which communicate by message passing. OTP design principles primarily derive from the need for reliable fault tolerant systems and the 'Let it crash!' philosophy, which is an approach to error handling that seeks to preserve the integrity and reliability of a system by intentionally allowing certain faults to go unhandled on a process level. (See e.g. [54], [55])

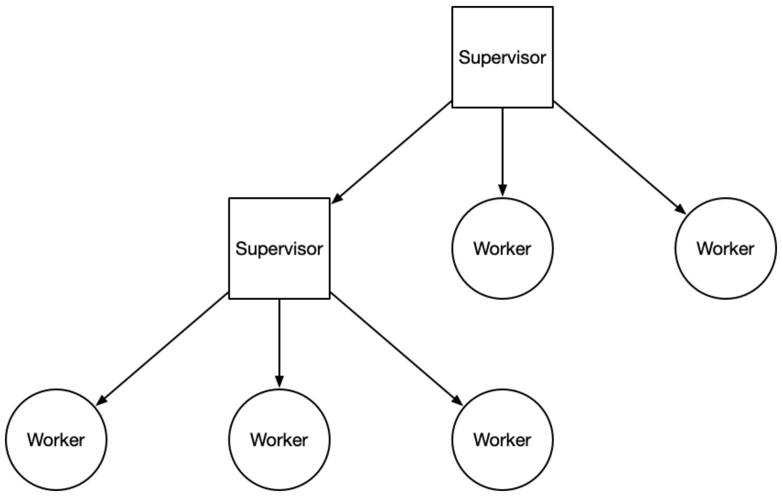

Figure 5: Erlang Supervisor Tree

The Erlang runtime system (ERTS) is made up of a language virtual machine (see 2.6) running on top of the host operating system or even 'bare metal' [24], a kernel providing low-level services such as distribution and I/O handling, and a standard library. The standard library includes base modules which reflect on the design patterns used in OTP: process supervisors, servers, state machines and generic event handlers. A process supervisor is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is to keep its child processes alive by restarting them when necessary according to a configurable strategy. Supervisors are managed in a hierarchical tree of processes and usually every Erlang process is part of that tree. Further OTP provides the user with a way to package system components based on a concept called application and such applications are automatically hooked into the supervisor tree when started. The ERTS does allow hot code reloading, e.g. it is possible to reload program code while the system is running [56].

3.4 TrustNode Platform

The TrustNode basic architecture has three main components:

- FPGA containing the packet stream processor "FlowEngine"
- Control processor
- High speed Ethernet Interfaces.

The Ethernet interfaces are connected to the FPGA. The idea is that most of the traffic is forwarded from port to port by the FPGA and only a small part handled by the control processor. The directly forwarded traffic has very low latency and offloads load from the processor. The connection between FlowEngine FPGA and control processor has two logical interfaces:

- Data interface for packet transfers
- Control interface for FlowEngine configuration.

The FPGA contains 3 methods to detect offloading traffic:

- for Ethernet switching a Media Access Control (MAC) table of learned addresses
- for SDN a Flow Cache which is programmed by the control processor
- a hardwired, hierarchical routing algorithm for IP Version 6 (IPv6).

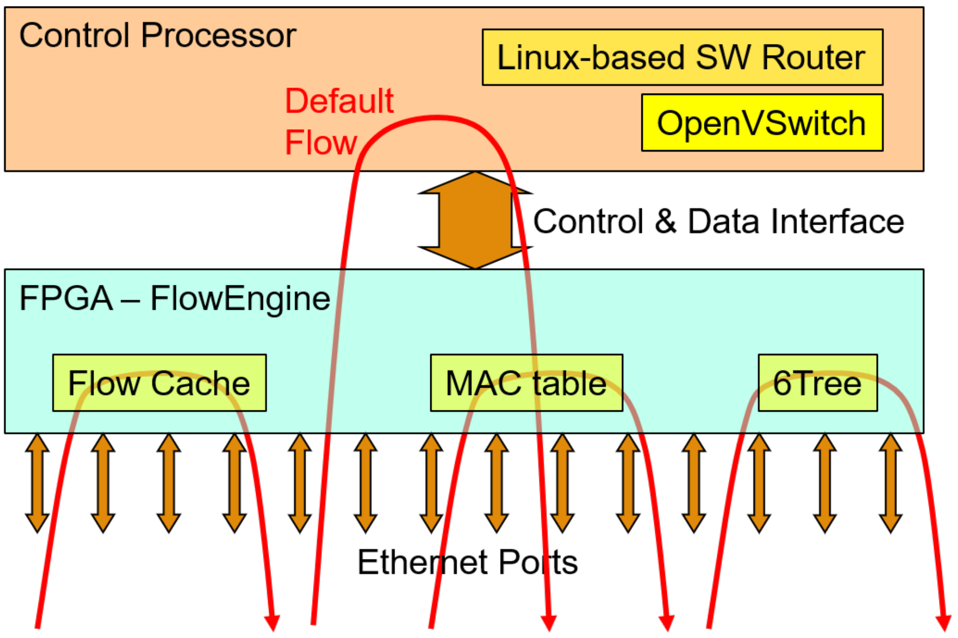

Figure 6: InnoRoute Generic Node Architecture

The first implementation of the architecture was the TrustNode with 12 Gigabit Ethernet ports, a FPGA type Artix 200T and a x86 ATOM processor with 4 Cores. Connection between FPGA and ATOM is a Peripheral Component Interconnect Express (PCIe) Interface with 4 lanes and a maximum transmission speed of 20Gb/s for both data and control interface. The second implementation of the architecture has 3 Gigabit Ethernet interfaces, a FPGA type Artix 35T and a Raspberry Pi as control processor. In this case the control interface is realized as Serial Peripheral Interface (SPI) bus in the Raspberry General Purpose Input/Output (GPIO). The FlowEngine is scalable in throughput, number of ports and other parameters such as number of flows, number of Flow Cache entries etc. Therefore, the FlowEngine for the Real-Time HAT has been derived easily from the TrustNode.

TrustNode and FlowEngine have been published in [57].

3.5 Trusted In-Network Computing (TINC) Platform

Trusted In-Network Computing Platform (TINC) is a distributed function orchestration platform for edge-computing applications. It is built on the concepts of secure containers, Function-as-a-Service (FaaS) and auction-based resource allocation.

The platform enables confidential computing at the edge by providing secure offloading of data processing and storing for distributed containers using hardware level security provided by Intel SGX [40]. The capabilities of secure computing are exposed through the secure containers framework (SCONE - [58]) that supports a greater variety of programming languages (i.e. applications can be written in various programming languages but still run inside secure Intel SGX enclaves) and offers a richer set of features and tools. Executing applications inside secure enclaves enables the protection of sensitive code and data from tampering by unauthorized entities such as external adversaries or even privileged software such as the hypervisor, the operating system and the Docker engine.

In addition to the shielded execution, the platform provides remote attestation and thus applications can only retain secrets (keys, configuration files, environment variables and arguments) only if they have have been remotely attested by a trusted managed service. Applications running in secure enclaves can communicate using TLS channels with other enclaves once they have been mutually attested as well (i.e. proved their trustworthiness to one another).

Application requirements are described using metadata documents called "Manifests" and are distributed to TINC nodes in order to proceed with the allocation of resources. The manifest document is using JSON as a description notation.

In addition, the platform does resource allocation through auction mechanisms [59]. For the platform, resources are considered as a pool of interconnected virtualized hardware offered via independent marketplaces located at each node. Constrained resources (e.g. containers) at the edge are being auctioned and provided to bidding applications offered by application providers. A distinct mechanism of the system is the ability to offer resources to distant markets that are over-utilized and compensated by that. Creating markets where resources are leased by potential bidders provides two significant benefits:

1. Infrastructure Providers are introduced to new profit opportunities since offering their resources to distant markets produces new revenue streams for them.
2. Application Providers enhance the QoS of their customers.

The main building blocks of the TINC platform are presented in Figure 7.

The TINC engine combines components that contribute to resource allocation, manifest parsing, requirements fulfilment and collection of reports from the containers running applications. The application life-cycle is bound to the Observer component which manages the execution workflow of the application and runs inside an enclave. The Observer and the application are required to be locally attested prior to the application's business logic execution. Finally, the Observer advertises usage

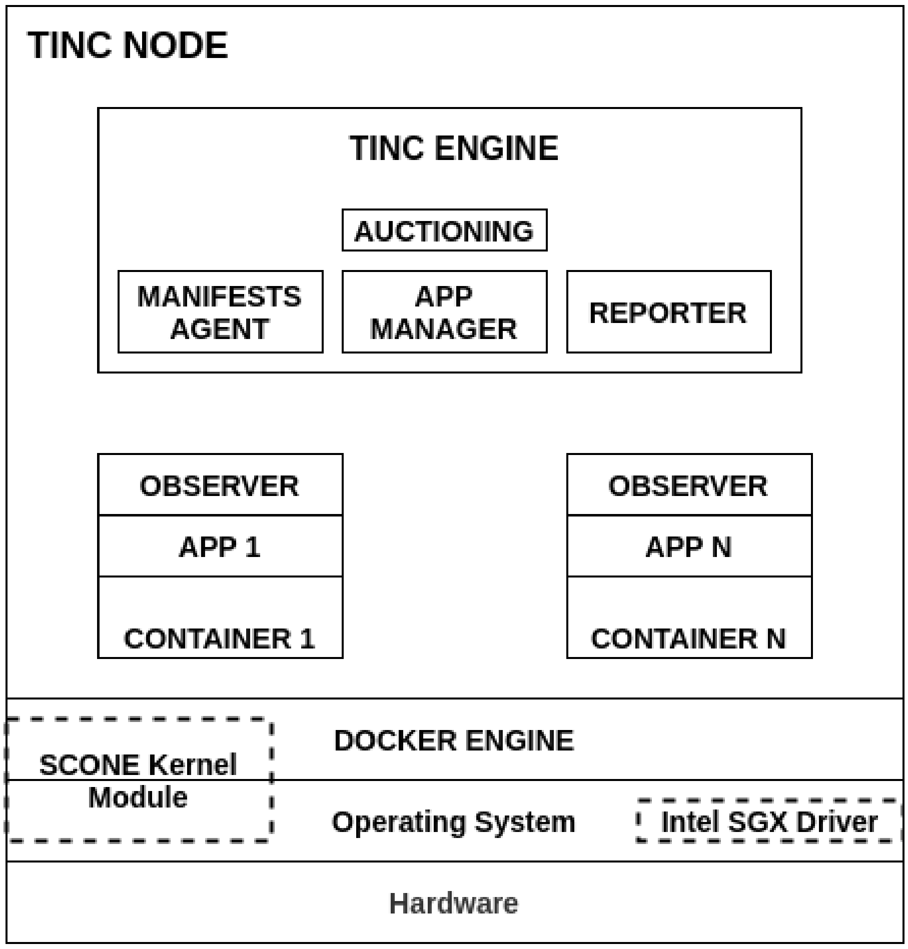

Figure 7: Fluentic TINC node building blocks

information to the TINC engine that facilitates features like auto-scaling and failure recovery.

3.6 Compute First Networking

CFN [60] is a distributed computing environment that provides a general-purpose programming platform with support for both stateless functions and stateful actors. CFN can lay out compute graphs over the available computing platforms in a network to perform flexible load management and performance optimizations, taking into account function/actor location and data location, as well as platform load and network performance.

CFN is using Python as a programming environment provides CFN-specific decorators for declaring actors and different types of actor methods that can be invoked remotely, as depicted by the example in figure 8.

Developers designate functions as referentially transparent or referentially opaque through these decorations. CFN defines wrappers around decorated classes and functions allowing to perform additional processing. On invocation, each decorated function immediately returns a future to the results instead

```
@cfn.actor
class Simulator:
    #some state
    @cfn.transparent
    def __init__(self):

    #set range based on internal state
    @cfn.transparent
    def setRange():
        #compute range here
        return custom_range

    @cfn.transparent
    def run(seed):

@cfn.opaque
def getRandom(range_future):
    range_data = cfn.get(range_future)
    return random.randint(range_data)


sim = Simulator.remote()
r_future = sim.setRange()
seed = getRandom(r_future)
result = runSimulation(seed)
```

Figure 8: CFN Python Program Example

of the actual data. The future can be then used either in a subsequent call to retrieve the result or passed as an input parameter to other functions. In order to retrieve the computed data, functions call get(future); this will block until the requested data has been computed and fetched.

In CFN, each class is a stateful actor comprising all its methods. CFN automatically recognizes state modifications and is able to migrate actor's state among different workers. The whole process is transparent for developers. Decorating an actor's functions does not require a different approach from that used for stateless functions.

We describe the distributed computing aspects of CFN in Piccolo deliverable D3.1 [2].

4 Design Goals and Research Questions

In this section we propose the design goals for a Piccolo node and highlight the research questions raised. These goals are informed by the requirements arising from the use-cases in deliverable D1.1 [1] and the distributed computing abstractions described in deliverable D3.1 [2].

4.1 Multi-tenancy

A highly desirable property for a Piccolo system is to enable computation from multiple users to be distributed across the nodes. Mechanisms will be required to allocate resources to specific users and to provide isolation guarantees so one tenant cannot interfere with the operation of another tenant. The protection must extend from application code (including an Execution Environment (EE)) to application data and any communication (via the network for example). Security considerations mandate that Piccolo should include robust authentication and access control mechanisms but these alone will not suffice. We propose to examine the impact of using different technologies for allocating resources to workloads whilst providing a high-degree of isolation. We expect that different approaches may be required for different examples of Piccolo nodes and we will also take into account the fact that some workloads may be more sensitive than others with respect to their requirements for isolation. In addition, sharing of resources may also require workloads to be prioritized especially if over-commitment is possible. Piccolo will consider prioritization mechanisms in order to manage the allocations of resources based on the demand of each workload and reduce any under-utilisation.

4.2 Security

Security considerations will shape the design of the Piccolo nodes in several ways. A Piccolo system may be constructed in many different ways including using on-premise nodes within a private network, using shared nodes within a public network and even using both private and public nodes. Particular issues already identified include:

- The lifecycle of a Piccolo node: how nodes are configured, booted and how they are incorporated into a Piccolo system.
- Access controls to determine who can manage the Piccolo system and nodes.
- Authentication for users or applications requesting access to Piccolo resources.
- Enforcement of on-node resource allocation between users, execution environments and Piccolo functions.
- Isolation between users - implementing a layered approach to safeguard information: not just application code and data but higher-level information about other users.

- Secure application updates: Applications require to be updated continuously in order to add new features or patch security vulnerabilities. It is therefore crucial for Piccolo nodes to ensure that new versions are genuine prior to transferring any secrets from the previous version of the application.
- Rollback protection: While cryptography can preserve confidentiality and integrity of data via encryption mechanisms, it does not protect for the class of rollback attacks by which an adversary attempts to replace the current state of the file system with a previous version. However, the prevention of rollbacks entails serious runtime overheads and application re-engineering, something that needs to be addressed in Piccolo.
- The heterogeneity of Piccolo nodes, from powerful server machines to single-board computers, dictates for careful decisions in terms of security mechanisms and authentication strategies. Security approaches that require expensive hardware or have large memory footprint will be infeasible except for a subset of Piccolo nodes.

4.3 Multi-Platform Operation

Different use-cases described in Piccolo deliverable D1.1 [1] demands support for different requirements from underlying hardware infrastructure (from competent nodes to FPGAs), support for hardware accelerations (GPU-enabled computations to secure enclaves) and operational environments. Furthermore, it is reasonable to assume that microservices from different target use-cases will operate on common Piccolo infrastructure, each necessitating the execution environment best capable of supporting its operational requirements. Therefore, it is pertinent that Piccolo, by design, should be able to flexibly support the heterogeneity in deployment infrastructure and wide-array of virtualization solutions. This, however, raises several research considerations.

Firstly, within Piccolo's context, different hardware devices may lend support for different levels of virtualization. While more resource-rich hardware in the infrastructure can enable the operation of containerized (e.g. Docker, LXC) applications, the network will also include other heterogeneous hardware that may support other variety of virtualizations (e.g. unikernels, Erlang VMs). Piccolo's primary objective will remain to support the wide-array of virtualization support in different hardware levels available in the infrastructure as flexibly as possible. The challenge is not only restricted to discovering the most optimal hardware in the resource pool for servicing the application (based on the virtualization used), but also to effectively utilizing resources that can concurrently support more than one virtualization platform. In such cases, the Piccolo orchestration "agent" must integrate underlying virtualization technologies available on every device via extensible system calls that allow dynamic changes in the service (actor) operation, e.g. instantiating, scaling in/out, restarting deployment etc. Due to the heterogeneous nature of hardware in the infrastructure (in terms of capabilities and platform support), Piccolo's agent design must depart from a traditional monolithic software block but instead incoporate a more *plug-in* like approach that adapts execution environment support based on the underlying hardware configuration.

Secondly, different services may utilize execution environments that are closely tied with the associated hardware-level support for optimal operation. For example, let's consider the automated driving use-case described in Piccolo deliverable D1.1 [1]. In this case, the vehicle-facing microservice takes the camera-feed captured by the in-vehicle camera and denatures private information (such as faces, objects or scenes) from the video stream. Due to the privacy-centric nature of the microservice's workload, it may utilise TINC platform (discussed in Section 3.5) that operates only within the TEE boundaries (over technologies like Intel SGX, AMD SEV etc.) using secure containers. Further, the actor may also require support from hardware accelerators/GPUs to fast-track its operation. In such a case, not only should Piccolo be able to support such hardware dependencies for individual microservices, but it must also allow simultaneous execution of other actors on the same hardware that do not impose such dependencies.

4.4 Multi-Protocol

Piccolo aims at developing a practical platform that can be used for implementing different applications leveraging the different node platforms mentioned above. Some of the relevant applications such as Sensing Feeling's Vision Processing are using containers as computation modules and RESTful HTTP-based protocols for inter-function communication.

While the Piccolo system will support such legacy systems, it will also support other platforms (with other specific protocols) and develop a new protocol that will enable future Piccolo applications. Piccolo deliverable D3.1 [2] will present a set of existing protocols and initial invariants for the design of a new protocol.

A Piccolo node (through an agent API) would map local API calls (for example, for Remote Method Invocation) to specific protocols.

4.5 Automatic Resource and Failure Management

The Piccolo Platform strives to provide features such as EE agnostic mechanisms for automatic resource and failure management of functions. Resources in general include compute (CPU, (GP)GPU), storage (memory, database, etc.) and data received from networks or even access from other EEs (local or remote) via well defined interfaces. Those interfaces will support features which are listed below:

For example, the **handling of shared memory** in the light of cooperative execution between multiple functions is challenging across EE boundaries as well as across a group of (local) functions. Lifetime requirements of allocated memory might differ between cooperative functions challenging the balanced memory utilisation on a Piccolo platform. Furthermore, function composition (or function chaining) introduces a need for **sharing/migrating function state** between instances which might be running on another Piccolo node. Concepts such as **caching** of function results or storing function state for subsequent functions improves the overall performance of function execution on Piccolo node by increasing the availability of data, e.g., in case of a function failure. Mechanisms for close (real-time) **monitoring of resource consumption** helps to vertically scale the environment (e.g. more MIPS) in an automated way.

While mechanisms to access resource management strive for performance improvements, it raises security concerns on many levels. **Security mechanisms** and support of **multi-tenancy** are required to ensure the integrity of the callee and the called function which might be automatically instantiated from a remote source. Another concern is the validation of in- and output data operated by functions.

As it is expected that a Piccolo node executes functions from different parties, efficient **task scheduling of functions** describes a crucial feature, for example, the possibility to preempt the execution of a function in favour of a function with higher priority. Furthermore, such mechanisms need to **handle failed executions of functions** as well. Since such a failure can occur on very different levels of a Piccolo node or EE, automatic resource and failure management mechanisms need to support different types of strategies to resolve the failure such as rollback of failed state updates, restart functions/EE etc.

4.6 Location-independent Naming and Discoverability

A Piccolo system can be a highly dynamic system:

- nodes can fail (and by transitions, the functions running on them);
- nodes can be added or allocated dynamically (for example edge computing nodes in a mobile automotive scenario);
- functions (services) can be instantiated on nodes dynamically, for example as per the dynamic execution of programmes, or for implementing high availability (redundancy) or scalability (goals).

In Piccolo, it should be possible to integrate new nodes into the set of available compute nodes in an infrastructure. For discovering and allocating nodes dynamically, it is important that the system does not require any static, possibly pre-configured, knowledge of node addresses.

One domain which benefits from such design are connected vehicles. In today's connected vehicle solutions, nodes are able to communicate with each other in an ad-hoc fashion by choosing from a variety of wireless networking technologies such as cellular or Dedicated Short Range Communication (DSRC). As a result, it enables establishing communication between vehicles and other network participants. To benefit from this, additional discovery mechanisms are required. WiFi based solutions like IEEE 1609 [61], or ETSI ITS G5 [62] broadcasting dedicated discovery messages to identify communication partners while cellular solutions like LTE-V [63] uses the base station as proxy.

Functions may also be instantiated (and removed) continuously on such a dynamic set of nodes. In such a system, function invocation, access to data etc. most be strictly location-independent:

- application and their functions will be developed in an SDK that is unaware of deployment configuration (locations, addresses);
- runtime, nodes and functions can be instantiated dynamically, hence all aspects of function interaction and node management must be location-independent.

As a design goal, Piccolo will thus have location-independent naming of nodes and functions.

4.7 Capacity, state and performance specification of a node

In cloud technologies, the capacity of compute platform resources assigned to a client is normally specified as base profile of

- CPU cores
- RAM
- Disk storage

This is not an abstract specification and has little intrinsic mathematical basis. The actual throughput realised by a client load is dependent on the specific details of the hardware.

The actual throughput of a client load on a CPU core depends on the instruction set architecture (ISA) (eg x86_64, ARMv8, etc), the clock rate (which may be modified to control heat dissipation), the size of different levels of cache, the availability of advanced instructions (eg vector arithmetic), etc.

RAM is different to disk storage only in access time but the exact access times are not specified by simply saying RAM or disk, and aspects such as Non-uniform Memory Access (NUMA) and Solid-State Drive (SSD) v spinning disk can influence the outcome.

In addition, these basic cloud parameters do not give any specification to the I/O capacity. For most cloud workloads this is not normally relevant as with these workloads, most practical servers tend to be compute resource bound and not bound by I/O capacity. In short, the specification of resources in cloud technologies is empirical and pragmatic and oriented to the majority workloads which are hosted.

NFV has already encountered issues with this pragmatic specification as many NFV workloads are I/O bound and not compute bound. Indeed, a significant part of enabling NFV as a technology was including extra hardware and software technology to remove unnecessary bottlenecks in I/O performance which were not relevant in the support of most cloud workloads. These technologies included

SR-IOV and DPDK and NFV has worked on adding additional parameters to the specification of host compute capability to include the explicit specification of these technologies.

A further feature of cloud technologies including both virtual machine (VM) services and container services is that the code which is executed is binary code which must be compatible with the ISA. Therefore, a dependency on the ISA is unavoidable and gives validity to that aspect of the pragmatic specification.

However, in Piccolo, the objective is to be able to support the same function on different compute nodes including with different CPUs with different ISAs. The runtimes which support this are well established including JRE, Erlang runtime, WASM, Python runtime, etc, however, it is not normal to specify throughput capacity at the level of these runtimes. Indeed, there is no obvious parameters which would specify the throughput capacity of the runtime. Even if it were easy to specify the throughput capacity of a runtime, this would not solve the problem for Piccolo as the capacity measure would be different for each runtime.

In Piccolo, we propose to proceed with two parallel approaches.

- Characterise throughput as perceived by the function and in terms of meaningful to the function. This effectively measures the achieved throughput rather than trying to predict the throughput. This is like the technique assumed in cloud native services, for example, based on Kubernetes which scales the resources to accommodate demand on the workload.
- Work on a more mathematic way for defining capacity as perceived by the function and couple this with a more mathematical way of mapping this capacity to the runtime and hardware resources.

5 Initial Piccolo Node architecture

5.1 System overview

We now introduce the main constituents of a Piccolo Node, which we will do incrementally. As discussed above, the generic Piccolo Node will be a hardware platform, possibly including special-purpose hardware, running an operating system and/or virtual machine monitor (VMM) and provide an execution environment for Piccolo functions. One entity running on a Piccolo Node is the Piccolo Agent that interfaces the node and its capabilities to the (other nodes in the) Piccolo System. Together these elements make up a minimum – simple – Piccolo Node. Figure 9 shows the conceptual view of such a simple Piccolo node. Note that the elements depicted are – throughout this section – logical components and do not imply any specific way of implementation.

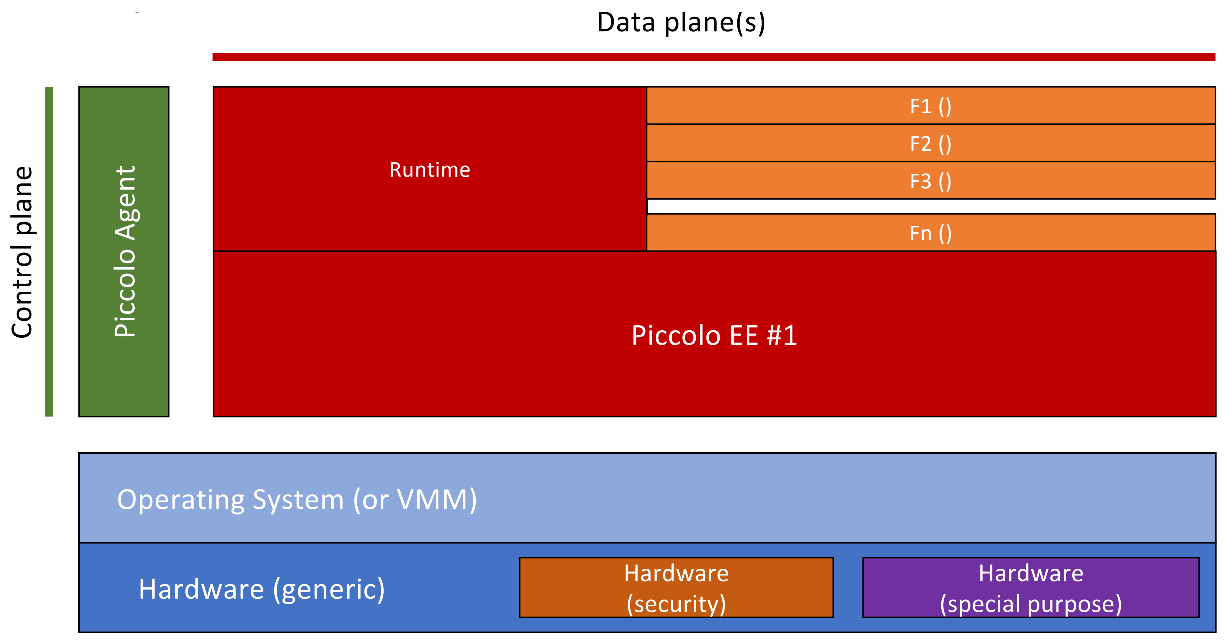

Figure 9: Simple Piccolo Node

The individual constituents are described in more detail below before we refine and extend the composition to capture more detail:

- The hardware itself is considered general purpose computing hardware (e.g., a rack-mount server, generic PC, or single-board computer such as a Raspberry Pi). The hardware may provide, inherently or via add-on modules, specific security functionality (e.g., TPMs) and/or a special-purpose accelerators (such as GPUs, APUs). The Piccolo hardware platforms may also be constrained (embedded) devices (e.g., a microcontroller). A common requirement is that the hardware is capable of executing at least the functions of the following three items.
- A (standard) operating system or hypervisor/virtual machine manager will provide the low-level hardware abstraction and node-internal management functionality for isolation, resource

management, etc. upon which Piccolo functionality can be built.

- A Piccolo EE provides a runtime environment that Piccolo functions are executed within. The figure shows a split between EE and runtime, which is to be understood as follows: the runtime offers the specific (isolated) interface to the individual Piccolo functions (see below) once they are instantiated, whereas the EE groups all functions together, supports their execution and termination, and also selective interactions across functions.

- Piccolo functions can be executed within the EE and form a distributed data-plane via connections to other Piccolo nodes. We use the term "Piccolo function" in a broad sense to refer to any piece of executable code running on a Piccolo Node for the purpose of providing functionality within the distributed data plane. Such a function may be stateful or stateless, long- or short-lived, and exhibit different properties along further dimensions. Piccolo functions may be used to implement stateless lambda functions, actors, stateful servers, and other patterns. Note that Piccolo functions may use different data planes to interact with one another; see 5.4 below.

- The Piccolo Agent manages the EE and provides the control plane API to other Piccolo Nodes. The control plane defines the protocol interface across Piccolo Nodes for the purpose of managing Piccolo Nodes, exchanging management- and resource-related information among Piccolo Nodes, and discovering nodes and node capabilities, among other control and management functions, as we will discuss in section 5.4 below.

We do not expect every embodiment of a Piccolo node to necessarily contain these elements as separate components (or even to contain all of them). For example, it would be possible to build a Piccolo node in which the there is no real operating system and where the Piccolo Agent is part of a single EE.

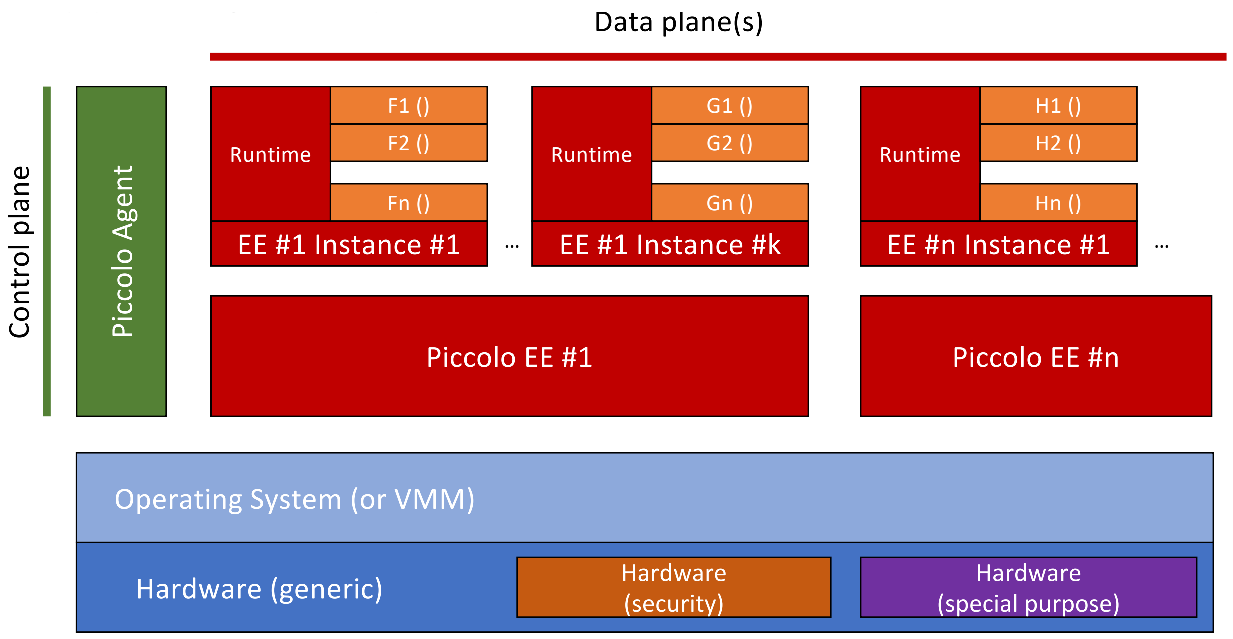

Figure 10: Piccolo Node featuring multiple Execution Environments, with several instances each

Figure 10 shows that a more than one EE may be provided (to allow Piccolo functions implemented using different language/runtimes for example), all of which would be under the control of the Piccolo Agent. A given execution environment (e.g., EE #1 in the figure) may be instantiated multiple times, e.g., with different access permissions to local resources and/or for different tenants. Node-local information exchanged may be supported in a controlled fashion among Piccolo functions within a single EE instance, across EE instances within a single EE, and across different EEs.

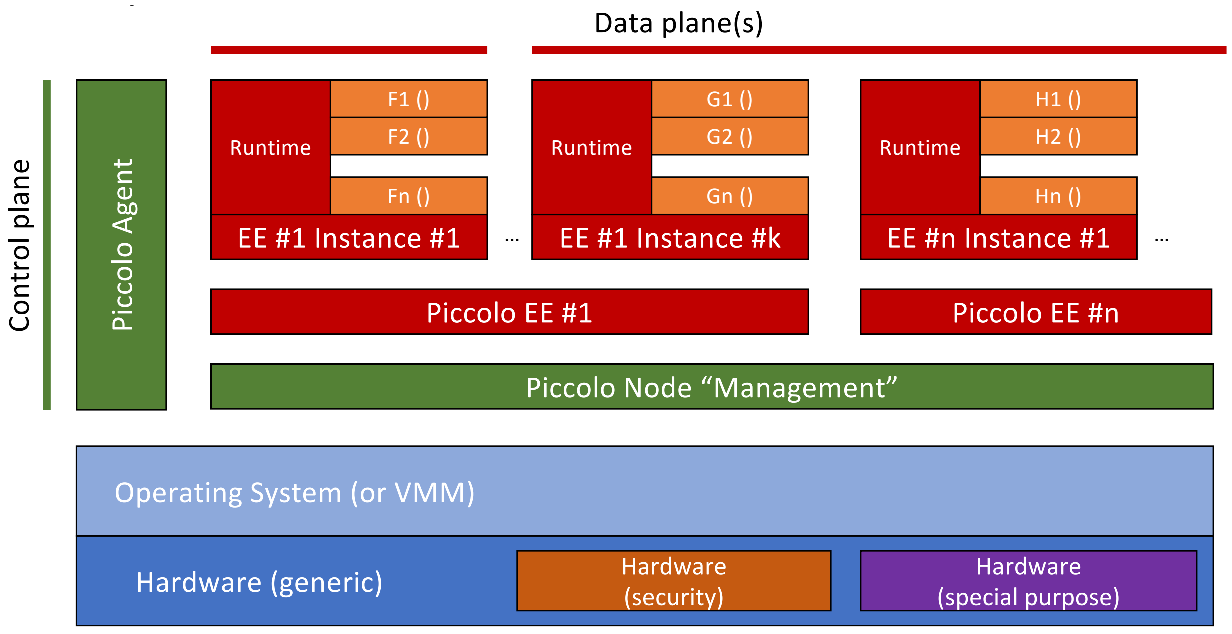

Figure 11: Piccolo Node with logical resource access management

Sharing of specialised hardware, such as for security or acceleration, may require a Piccolo Node management software component to manage access to the hardware, as depicted in Figure 11. Such software will be controlled by the Piccolo Agent and may be realized in different ways: leveraging OS-level access control and isolation features, using paravirtualization concepts, relying on access control and virtualization of the underlying hardware, or as an indirection layer via which all resource access must pass, among others. The implementation choice will be highly dependent on the underlying hardware and operating system platform, the specific protection goals to be achieved, performance requirements, and so on.

Finally, with the above mechanisms a Piccolo Node will naturally be able to support multi-tenancy, i.e., the execution of functions belonging to different users as well as the execution of the same function on behalf of different users. Controlled sharing of code and data may be provided.

5.2 Operation

The design of the Piccolo Node is flexible to allow different models of distributed computing to be realised with a network of connected Piccolo nodes. Below we define APIs and protocols that can be used to configure and control the capabilities of Piccolo nodes. This configuration could take place at boot-time only or it may be desirable to change the configuration on demand. In the very simplest implementation the node will provide a single EE within which a fixed set of Piccolo functions is available.

Every Piccolo Node is controlled by its Piccolo Agent as noted above. It interfaces to all the local components of the Piccolo Node to instruct them what to do and to retrieve state information from them. It interacts with the node operating system and, as necessary, the node hardware to ensure proper functioning of the Piccolo Node and ensure node integrity. The Piccolo Agent is the software running on the node that other machines communicate with using the Piccolo control plane.

The Piccolo agent provides several capabilities:

1. **Starting** a Piccolo function: the agent uses the API to communicate with the execution environment in which the function is started. This is essentially a provisioning step. Once started, the function is then available to be executed.

2. **Querying**: Using the Piccolo API, a remote node can query the Piccolo agent to determine what Piccolo functions are available for execution. The remote node uses the Piccolo API to request to execute a particular function and the Piccolo agent provides the information required (an authorisation token and namespace Uniform Resource Locator (URL) for example) for the remote node to use to trigger the actual execution.

3. **Executing**: Using the information provided by the query stage, the remote node triggers the execution of the Piccolo function, usually via direct communication with the execution environment (although an indirection via the Piccolo agent is also possible)

4. **Discovering**: The Piccolo agent monitors the local network and is aware of other local Piccolo nodes. This information may be stored and can be queried by other Piccolo nodes. In this way more capable nodes can provide information to less capable nodes about function availability. Discovery may extend beyond nodes colocated inside a single broadcast domain network via dedicated links (cf. dedicated routers for forwarding traffic to a local network), thus forming an overlay. Maintenance of such an overlay is for future study.

To realise these functions, the Piccolo Node uses a number of (conceptual) APIs for node-local interaction and protocols for inter-node communication. We summarise these interfaces in Figure 12 and show how the various components connect together: the local APIs are indicated by numeric (1–11), the protocol APIs by alphabetic references (A–C). In the following two sections, we first describe the abstract APIs and then the abstract protocols.

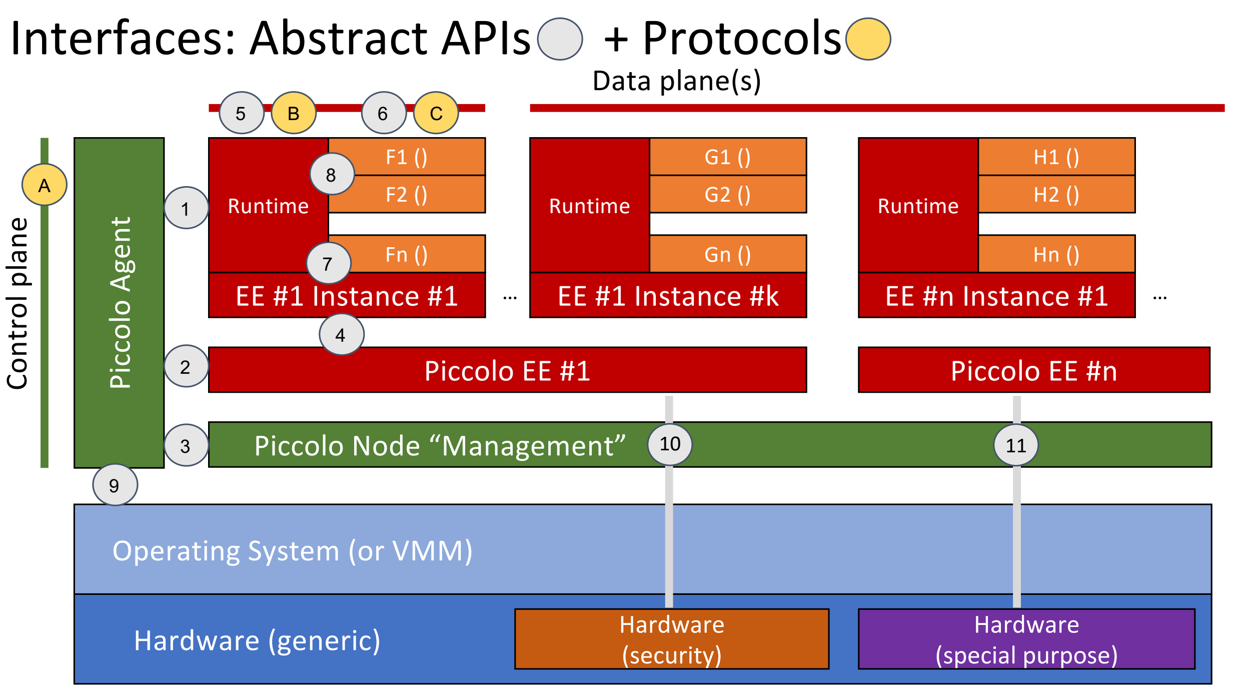

Figure 12: Abstract APIs and Protocols

5.3 Abstract API

As shown in Figure 12, Piccolo identifies 11 abstract APIs between the individual components. Note that these APIs do not imply that a dedicated library or other function call or message passing interface exists; the APIs are logical in nature, i.e., they may be realized as depicted but they may also be grouped together to match implementation needs, may not exist visibly as APIs if components are integrated, may be part of OS- or other systems-level APIs, among other variations.

In the following, we describe these logical APIs and their intended abstract functionality. Future deliverables will address some of these in more detail, while others may remain implementation-specific (or: proprietary) and thus may not require a detailed specification.

The APIs are structured into four groups: (1)–(4) are about configuring and managing EEs and EE instances/runtimes; (5)–(6) concern the interaction of runtimes and functions with the local networking stack (specific to a given data plane); (7)–(8) provide the instantiation of and interaction with and across functions; and (9)–(11) are about the interaction with local resources via the operating system.

1. **Agent–Runtime**: This APIs keeps the Piccolo Agent informed about the currently running functions. This contributes to the the complete view of the state of the Piccolo Node.

    - Function registration for discovery: an EE/runtime will notify the Piccolo agent about currently running functions and their static and dynamic properties; static properties include the function names and their capabilities, possibly also their present load and resource consumption. This information allows a Piccolo Agent to advertise available functions to other Piccolo Nodes.

- Query routing: the Piccolo Agent may issue or relay requests about running functions. Queries allow the control plane to collect specific state information independent of the Piccolo Agent.

2. **Agent–Execution environment**: The main purpose of this API is that the Piccolo Agent obtains an accurate and up-to-date view of the resources used and available in the Piccolo Node and enables the agent to manage which functions shall be ready to run. (The execution environment may itself decide when to schedule which function.)

   - Provision a function: the Piccolo Agent may install a function into an EE, thus making it available for execution. It may also remove a function. This essentially relates to (persistent) storage management.

   - Instantiate, terminate: the Piccolo Agent may instruct an EE to start and stop execution of a function. In contrast to provisioning, this is about active resource utilization management (i.e., "running process"). Note that it is up to a function in cooperation with the EE how to operate on incoming function calls, i.e., whether to queue the calls, whether to spawn new function instances, or whether to manage multiple calls within an instance.

   - Query capabilities: the Piccolo Agent may query an EE about its capabilities and (statically) available resources.

   - Query / notify about load: the Piccolo Agent can determine the current load of a given EE.

3. **Agent–Node management**: This API is used by the Piccolo Agent to manage which EEs may use which resources on the Piccolo Node.

   - Configure node resource accessibility: The Piccolo Agent may decide which resources of the node are accessible by the EE or EE instance; this is a qualitative decision in the sense of access control. These resources may include CPU cores, special-purpose hardware, persistent storage, network operations, and operating system calls. An open question is if the Agent should be able to exercise this control to the extent of which functions running inside an EE may access which resources.

   - Resource allocation to different EE: Beyond the basic permissions, the Piccolo Agent may also govern how much resources a given type an EE or EE instance may utilize; this is a quantitative decision in the sense of quota and prioritization.

4. **Execution environment to Execution Environment instance (plus runtime)**: This API controls the invocation of EEs and the configuration of their capabilities.

   - Create a new (or terminate) an EE instance: the EE "class", "framework", or "platform" may create new EE instances – on-demand or proactively – and terminate those when they are no longer needed. In a simple Piccolo Node, only one EE instance may exist, in which case the configured EE class constitutes also the only available instance and dynamic creation and termination of instances are not supported.

- System capabilities: The EE defines which system features a given EE instance is entitled to access and how many system resources are available to it. These permissions are prescribed by the Piccolo Agent.
- Local data exchange: Different execution environment instances may share data in a controlled fashion, which is realized via this part of the API, as is the controlled exchange with other EEs and their instances. Note, however, that calls between different functions of the same or different EEs logically pass across the networking stack and would thus, logically, exercise APIs (5) and (6), even though an implementation may decide to perform optimization along the lines of a loopback interface.

5. **Runtime–Network**: This API governs the control exchange for the runtime and its functions to the local networking stacks. This includes all functionality relevant to making functions accessible, creating, and possibly migrating them.
    - Register, deregister function "identifiers" with the network: Each function instantiated by a runtime must be reachable under its identifier(s) via the network and thus registered with the network stack; similarly, the function must be deregistered when no longer available. This (de)registration process is complementary to informing the Piccolo Agent because the latter is responsible for sharing metadata about the available functions in the control plane, while actual routing towards functions is performed directly across the network stack(s); see also API (6).
    - Accept, fetch function via the network: A function may need to be instantiated for which no code was locally provisioned before. This may be the case if the respective function is directly received from a network as part of an incoming message or needs to be fetched from a repository via the network. This task of retrieving function code and making the function available for (immediate) local instantiation is supported by this API.
    - Migrate (active) functions across the network: Functions may need to be suspended on one Piccolo Node, migrated to another node, and then resumed on the latter. This API supports marshalling code and state of a function and sending it across as well as receiving it from the network.
6. **Function–Network**: While API (5) is responsible for control interaction, this API takes care of the actual function invocation and/or data exchanges between two functions as part of the execution of a program logic. In the following, we use the term "connection" to refer to how data exchanges with other functions, abstracting away from the question how these messages are enveloped (e.g., connection-oriented vs. connection-less, transaction-based or not, reliable or not) and what sizes they may have—which may depend on the network architecture and the

function semantics, among other aspects.

- Accept incoming connections: a function is be able to passively receive and create/negotiate state for incoming messages.
- Create outgoing connections: a function can actively establish state for interaction with remote peers.
- Packet/message i/o: a function can send and receive data via the network, optionally including setting parameters for the data exchange.
- Terminate connections: a function can disband the shared state with a remote peer.
- Network failure handling: a function is notified about network errors and expected to react accordingly. Some intermediate abstraction may be provided (e.g., by a transport or session layer) to reduce the errors that become visible to the function.

7. **EE/Runtime–function**: The EE is responsible for managing the entire lifecycle of functions, including their execution, all handled via this API.

   - Instantiate function from local storage: An EE can create a running instance of a function; in Unix terms, this would be achieved by means of `vfork()` and `execve` system calls.
   - Replicate / clone a running function: An EE may create a replica of a running function and a function may clone itself (cf. Unix `fork()`). It is up to the implementation to determine if a function is supposed to manage its own replicas (maintaining oversight over which other replicas exist in a "root" function instance, or if this functionality is delegated to the EE.
   - Save state: A function may encapsulate its current state and save it to local storage. The EE may instruct a function to do so. This feature may be used for checkpointing and/or in preparation for suspending a function.
   - Retrieve state: A function may fetch its saved state from local storage. This may be used for recovery or to resume a function.
   - Suspend/resume function: A function may be suspended and resumed; resumption may happen on the same or a different Piccolo Node.
   - Terminate function: A function may terminate itself when done, or may be terminated by the EE (or a "root" function instance) when no longer needed.
   - Scheduling: The runtime is responsible for deciding which functions are to run when and for how long, possibly preempting the execution of other functions, similarly to (preemptive) scheduling in operating systems.

- Watchdog: The runtime may provide watchdog functionality to check the liveness of functions but also oversee their resource consumption. This provides the basis to administratively restart failed functions or terminate ones that have exhausted their resources.

8. **Runtime–function–function**: The EE and runtime support information exchange across functions.

   - Controlled data sharing: Functions may specify which other entities (e.g., based upon credentials, using appropriate security mechanisms) they want to share data with, and they may make data available to all functions. The precise mechanisms may vary between execution environments but will also include mechanisms for synchronizing data access,

   - Extend to sharing across EE instances and EEs: the above mechanisms can be extend via local means to sharing data beyond a single EE instance, across the entire Piccolo Node. Again, suitable protection mechanisms for controlled data sharing are required.

9. **Agent–environment**: This API allows the Piccolo Agent to assess the state of the local system in order to share system performance capabilities and the current load situation with other Piccolo Agents. This considers both static and dynamic parameters:

   - Static parameters include the available CPU cores, memory (possible per configured execution environment), storage capacity (both persistent and ephemeral), and the availability of special hardware resources (such as GPUs) and their characteristics. This also includes basic CPU parameters such as ISA and the OS interfaces available (ABI).

   - Dynamic parameters are essentially the current resource utilization, augmented by parameters that can estimate future load such as job queues, data expiry at different time horizons, among others.

   - In-between is information about the execution environments: which EEs can be executed at all (subject to available resources but also to node policies), which are available? Which could be instantiated?

   - Finally, the local node environment may take in information from other sources, e.g., routing protocols, local MIBs (or similar information, e.g., about interfaces).

10. **Agent/node management–secure hardware**: This interface to dedicated security hardware (such as TEEs, HSMs, etc.) governs the use of this specialized hardware by the Piccolo Node and thus also the (selective) availability to execution environments and Piccolo functions, such as the security hardware that could be used for the Piccolo control and data plane.

    - The interface may be – exclusively or not – used to protect Piccolo Node operation, i.e., the Piccolo Agent and its operation and its control of the node. The Piccolo Agent may include a trust code base to be executed only in a TEE and may make use of other secure hardware for identification and auditing purposes.

- At the same time, the interface may be used to protect EEs, functions, and also data. The split between untrusted and trusted parts of Piccolo functions is arbitrary: all code could be run in the untrusted domain at one extreme but also most code (except for the other APIs) could be run in the trusted environment. Functions would usually choose a middle ground.

11. **Agent/node management–special hardware**: This interface is similar to API (10) but about general dedicated hardware. This may include GPUs, APUs, FPGAs, and other accelerator hardware as well as sensors and actuators to be controlled by the Piccolo environment. The Piccolo Agent controls (via the node management) which hardware can be used by which execution environment. While the Piccolo Agent may also employ this hardware for control purposes (e.g., running ML algorithms for Piccolo resource optimisation), we expect the main use to be related to the data plane.

    - The Piccolo Agent manages controlled access to specialized h/w resources (e.g., GPUs) and offers this selectively to the EEs: for the EE, an individal instance, or even just specific functions.

    - The interface beyond access control is specific to the respective hardware resource and thus not covered in the high-level document.

5.4 Protocols

The APIs discussed above form the local interfaces between components inside a node. The protocol interfaces briefly outlined in this section cover the interactions across Piccolo Nodes. The main parts of protocol development are subject to WP3 and its Deliverables, at this stage especially D3.1, so that only a very brief outline is included here.

(a) **Agent Protocol (Piccolo API)**: This protocol interface is the common baseline for all nodes defined by Piccolo, allowing them to learn about each other and the respective node capabilities. It is the backbone of the Piccolo control plane. To this end, the Piccolo Agent Protocol supports the exchange of the following information:

- Node identity: Nodes learn about their identities and respective security parameters that can be used to authenticate nodes and validates messages from them. The identities may also serve management purposes.

- Properties: Nodes announce their local capabilities (runtime environments, resources, etc.).

- Load and other dynamic parameters may also be shared so that dynamic load balancing and prediction mechanisms may be implemented.

- Auctioning, payment: The protocol may be extended by additional protocols to support bidding for and then assigning ("selling") resources; this may including transferring payment references, but actual payment schemes and protocols are out of scope.
- Placement and resource allocation: A Piccolo controller may provide instructions to the Piccolo nodes that a certain function shall be executed on a given node and/or that certain resources shall be allocated to a given EE or function.

Semantically the protocol spans network architectures and may be mapped to different architectures.

(b) **Runtime protocols (Piccolo Runtime API)**: These protocols are specific to the runtimes and are used to exchange between Runtime instances. The protocols may be mapped to different network architectures, or the protocol may be specific to a given runtime. These protocols are used transfer the state of a function exported/imported from/into a Runtime so that (running) functions can migrate. As an option, function state may be represented independently of a speific Runtime to that functions may migrate across Runtimes. Some Runtime protocols may also offer multiplexing and dispatching functionality.

(c) **Function protocols**: Functions or function frameworks will define their own protocols (or define which standardized protocol stacks to use) and the semantics of the respective protocol elements. These are considered independent of the Runtime and the programming language the functions are implemented in. These protocols are used to represent function call, events, and other exchanges between functions.

6 Conclusion

In this deliverable report we have described the state of the art technologies that could be used in the implementation of a Piccolo node. We have described the hardware and software platforms currently used by the project partners. These platforms cover different points in the design space, which is appropriate as the Piccolo architecture is intended to enable heterogeneity in terms of compute and networking capability. This requirement for flexibility has been reflected in the design goals and has led to the initial node architecture described in this deliverable. Piccolo is a research project half-way through its first year and there are many questions still to be answered before the full realisation of the design goals can be achieved through the Piccolo Node architecture.

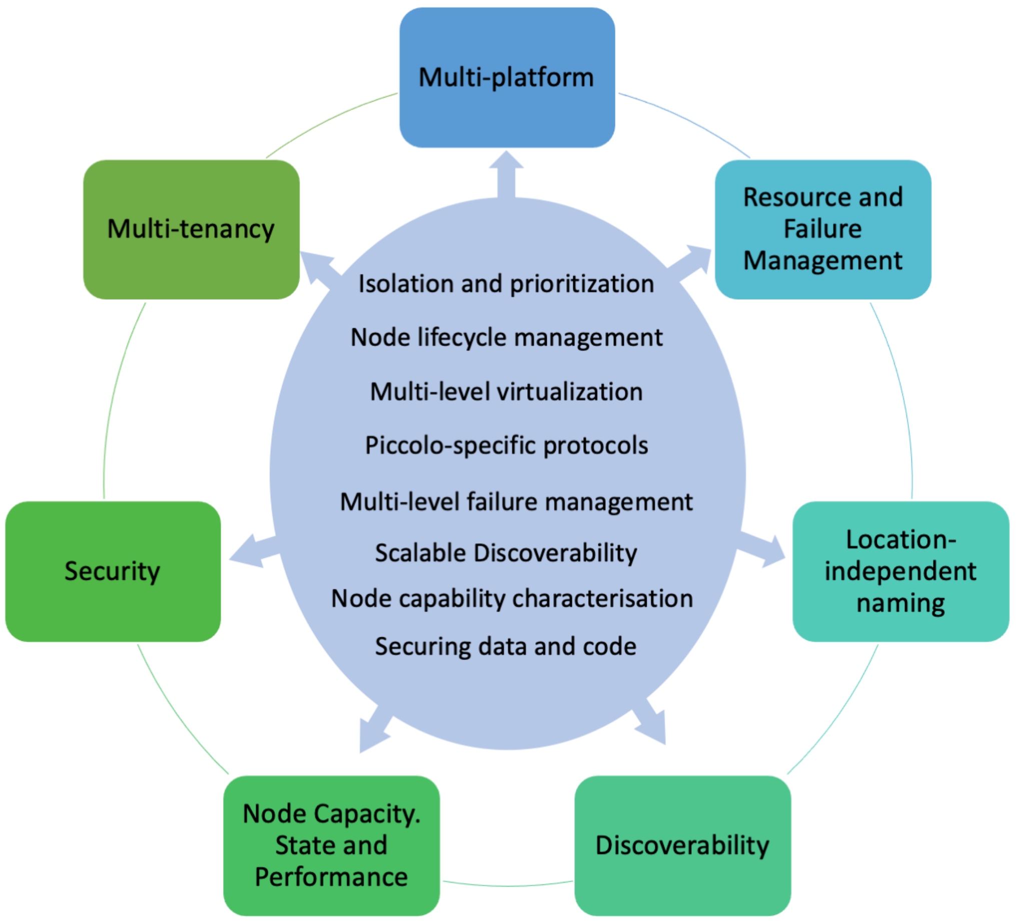
Multi-platform
Multi-tenancy
Resource and Failure Management
Isolation and prioritization
Node lifecycle management
Multi-level virtualization
Piccolo-specific protocols
Multi-level failure management
Scalable Discoverability
Node capability characterisation
Securing data and code
Security
Location-independent naming
Node Capacity. State and Performance
Discoverability

Figure 13: Piccolo Design Goals and Research questions

The next phase of work will include creating proof-of-concept implementations that use the abstract APIs and Protocols (an effort which stretches across WP2 for the Node and WP3 for the system) as well as research into questions about particular issues within the design. These issues will be explored as the work of refining the definitions of the APIs and Protocols continues and as the proof-of-concept implementations are developed and evaluated.

References

[1] Piccolo Project. *Use cases, Application Designs and Technical Requirements*. Deliverable D1.1. 2021.

[2] Piccolo Project. *Architectural Invariants for Distributed Computing*. Deliverable D3.1. 2021.

[3] *Docker*. `https://www.docker.com/`. Accessed: 2020-29-09.

[4] *Linux Conainers (LXC)*. `https://linuxcontainers.org/`. Accessed: 2020-29-09.

[5] Kata Containers Community. *Kata containers Description*. 2021. URL: `https://katacontainers.io` (visited on 03/03/2021).

[6] Vittorio Cozzolino. "Bridging the Gap: Orchestration and Resource Provisioning for Edge-Cloud Infrastructures". Unpublished Ph.D. dissertation. Technical University of Munich (TUM), 2021.

[7] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. "Exokernel: An operating system architecture for application-level resource management". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 251–266.

[8] Ian M. Leslie et al. "The design and implementation of an operating system to support distributed multimedia applications". In: *IEEE journal on selected areas in communications* 14.7 (1996), pp. 1280–1297.

[9] David Kaloper-Meršinjak et al. "Not-Quite-So-Broken {TLS}: Lessons in Re-Engineering a Security Protocol Specification and Implementation". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 223–238.

[10] Anil Madhavapeddy et al. "Unikernels: Library operating systems for the cloud". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 461–472.

[11] Anil Madhavapeddy and David J Scott. "Unikernels: Rise of the virtual library operating system". In: *Queue* 11.11 (2013), pp. 30–44.

[12] Anil Madhavapeddy et al. "Jitsu: Just-in-time summoning of unikernels". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 559–573.

[13] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[14] Rashid Mijumbi et al. "Network function virtualization: State-of-the-art and research challenges". In: *IEEE Communications surveys & tutorials* 18.1 (2015), pp. 236–262.

[15] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. "FADES: Fine-Grained Edge Offloading with Unikernels". In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (Los Angeles, CA, USA). HotConNet '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 36–41. ISBN: 9781450350587. DOI: `10.1145/3094405.3094412`. URL: `https://doi.org/10.1145/3094405.3094412`.

[16] V. Cozzolino et al. "ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment". In: *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. 2020, pp. 223–230. DOI: `10.1109/IoTDI49375.2020.00029`.

[17] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. “Edge Chaining Framework for Black Ice Road Fingerprinting”. In: *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking* (Dresden, Germany). EdgeSys ’19. New York, NY, USA: Association for Computing Machinery, 2019, 42–47. ISBN: 9781450362757. DOI: 10.1145/3301418.3313944. URL: https://doi.org/10.1145/3301418.3313944.

[18] *HaLVM*. https://galois.com/project/halvm/. Accessed: 2020-05-10.

[19] Alfred Bratterud et al. “IncludeOS: A minimal, resource efficient unikernel for cloud services”. In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2015, pp. 250–257.

[20] Joao Martins et al. “ClickOS and the art of network function virtualization”. In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 459–473.

[21] Vittorio Cozzolino et al. “UIDS: Unikernel-based Intrusion Detection System for the Internet of Things”. In: *DISS 2020-Workshop on Decentralized IoT Systems and Security*. 2020.

[22] Eddie Kohler et al. “The Click modular router”. In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[23] Avi Kivity et al. “OSv-optimizing the operating system for virtual machines”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 61–72.

[24] URL: https://grisp.org/ (visited on 03/03/2021).

[25] URL: https://rtems.org/ (visited on 03/03/2021).

[26] Pat Bosshart et al. “P4: Programming Protocol-Independent Packet Processors”. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (2014). DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[27] P4 Language Consortium. *P4 Compiler: The eBPF backend*. 2021. URL: https://github.com/p4lang/p4c/blob/master/backends/ebpf/README.md (visited on 03/08/2021).

[28] Eddie Kohler et al. “The Click Modular Router”. In: 18.3 (2000). ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: https://doi.org/10.1145/354871.354874.

[29] URL: https://www.dpdk.org/ (visited on 03/03/2021).

[30] URL: https://ebpf.io (visited on 03/03/2021).

[31] URL: https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html (visited on 03/03/2021).

[32] Huynh Tu Dang et al. “NetPaxos: Consensus at Network Speed”. In: New York, NY, USA: Association for Computing Machinery, 2015. ISBN: 9781450334518. DOI: 10.1145/2774993.2774999. URL: https://doi.org/10.1145/2774993.2774999.

[33] Xin Jin et al. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. In: New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 9781450350853. DOI: 10.1145/3132747.3132764. URL: https://doi.org/10.1145/3132747.3132764.

[34] Jan Rüth et al. “Towards In-Network Industrial Feedback Control”. In: New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450359085. DOI: 10.1145/3229591.3229592. URL: https://doi.org/10.1145/3229591.3229592.

[35] René Glebke et al. “Towards Executing Computer Vision Functionality on Programmable Network Devices”. In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450370004. DOI: 10.1145/3359993.3366646. URL: https://doi.org/10.1145/3359993.3366646.

[36] Thomas Kohler et al. “P4CEP: Towards In-Network Complex Event Processing”. In: New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450359085. DOI: 10.1145/3229591.3229593. URL: https://doi.org/10.1145/3229591.3229593.

[37] URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (visited on 03/05/2021).

[38] N. Asokan et al. “Mobile Trusted Computing”. In: *Proceedings of the IEEE* 102 (Aug. 2014), pp. 1189–1206. DOI: 10.1109/JPROC.2014.2332007.

[39] URL: https://www.niap-ccevs.org/MMO/PP/pp_skpp_hr_v1.03.pdf (visited on 03/05/2021).

[40] URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html (visited on 03/05/2021).

[41] Tu Dinh Ngoc et al. “Everything You Should Know About Intel SGX Performance on Virtualized Systems”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.1 (Mar. 2019). DOI: 10.1145/3322205.3311076. URL: https://doi.org/10.1145/3322205.3311076.

[42] URL: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (visited on 03/03/2021).

[43] URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c (visited on 03/03/2021).

[44] J. E. Smith and Ravi Nair. “The architecture of virtual machines”. In: *Computer* 38.5 (2005), pp. 32–38. DOI: 10.1109/MC.2005.173.

[45] Michal Cierniak et al. “The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment: Research Articles”. In: *Concurr. Comput.: Pract. Exper.* 17.5-6 (Apr. 2005), pp. 617–637. ISSN: 1532-0626.

[46] Tim Lindholm et al. *The Java Virtual Machine Specification. Java SE 15 Edition*. 2020. URL: https://docs.oracle.com/javase/specs/jvms/se15/jvms15.pdf.

[47] Andreas Haas et al. “Bringing the Web up to Speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 2017, pp. 185–200.

[48] URL: https://v8.dev/ (visited on 03/29/2021).

[49] URL: https://blog.cloudflare.com/introducing-cloudflare-workers/ (visited on 03/03/2021).

[50] *Erlang/OTP 23.2*. https://erlang.org/doc/index.html. Accessed: 2021-03-16.

[51] opcfoundation. *OPC Unified Architecture (UA)*. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[52] Florian Pethig. *Einfach anfangen mit dem Werkzeugkasten OPC UA*. 2017. URL: https://industrie40.vdma.org/documents/4214230/18583764/08%20Pethig_Werkzeugkasten%20OPC%20UA_1499339519885.pdf/66203259-98b5-415e-9b50-c6e7267368eb.

[53] gRPC Authors. *gRPC Project Website*. 2021. URL: https://grpc.io/ (visited on 03/26/2021).

[54] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

[55] Fred Hebert. *The Zen of Erlang*. https://ferd.ca/the-zen-of-erlang.html. Accessed: 2021-03-16.

[56] *OTP Design Principles User's Guide*. https://erlang.org/doc/design_principles/users_guide.html. Accessed: 2021-03-16.

[57] Chrstian Liss et al. "Architecture of a Synchronized Low-Latency Network Node Targeted to Research and Education". In: (2017). DOI: 10.1109/HPSR.2017.7968673.

[58] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[59] A. G. Tasiopoulos et al. "Edge-MAP: Auction Markets for Edge Resource Provisioning". In: *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 2018, pp. 14–22. DOI: 10.1109/WoWMoM.2018.8449792.

[60] Michał Król et al. "Compute First Networking: Distributed Computing Meets ICN". In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450369701. DOI: 10.1145/3357150.3357395. URL: https://doi.org/10.1145/3357150.3357395.

[61] "IEEE Guide for Wireless Access in Vehicular Environments (WAVE) - Architecture". In: *IEEE Std 1609.0-2013* (2014), pp. 1–78. DOI: 10.1109/IEEESTD.2014.6755433.

[62] European Telecommunications Standards Institute. *ETSI TS 102 636-3: Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetwork; Part 3: Network architecture v1.1.1*. Tech. rep. 2010.

[63] 3rd Generation Partnership Project. *RP-161298: LTE-based V2X Services*. Tech. rep. Sept. 2016. URL: https://portal.3gpp.org/ngppapp/CreateTDoc.aspx?mode=view&contributionUid=RP-161298 (visited on 05/13/2020).