# Deliverable D3.1

# Architectural Invariants for Distributed Computing and Technical Requirements

| | |
|---|---|
| Editor: | Dirk Kutscher – UEMDEN |
| 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: | 68 |
| Keywords: | Architecture, Distributed Computing, Protocols |

***Abstract***

This document describes key concepts, relevant state of art, and an initial set of technical invariants for the Piccolo system architecture.

## 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] WP3. Infrastructure
[Number and title of task]
[Document title] D3.1 Architectural Invariants for Distributed Computing
[Editor: Name, company]
[Work-package leader: Name, company] Dirk Kutscher, UEMDEN

## Copyright notice

# Executive Summary

Piccolo's aim is to develop a new type of flexible distributed computing framework, and to apply this to a set of relevant application scenarios.

Piccolo's Work Package 3 (Infrastructure) is concerned with defining the overall system architecture and the elements of the distributed computing infrastructure such as protocols and distributed algorithms.

In this deliverable D3.1 "Architectural Invariants for Distributed Computing and Technical Requirements" we present key architecture invariants and describe Piccolo's relationship to seminal key concepts and state of the art.

## List of Authors

| **Company** | **Author** |
|---|---|
| ARM Ltd. | Chris Adeniyi-Jones |
| British Telecommunications plc | Adam Broadbent, 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, Uthra Ambalavanan |
| Sensing Feeling | Dan Browning, Jag Minhas, Chris Stevens |
| Stritzinger GmbH | Mirjam Friesen, Sascha Kattelmann, Peer Stritzinger, Stefan Timm |
| Technical University Munich | Jörg Ott, Raphael Hetzel, Nitinder Mohan |
| University of Applied Science Emden/Leer | Dirk Kutscher, Laura al Wardani |

# Table of Contents

# List of Figures

# Abbreviations

| | |
|---|---|
| **ASIC** | Application-Specific Integrated Circuit |
| **CNCF** | Cloud Native Computing Foundation |
| **CPU** | Central Processing Unit |
| **CRDT** | Conflict-free replicated data type |
| **ETSI** | European Telecommunications Standards Institute |
| **FaaS** | Function as a Service |
| **FPGA** | Field Programmable Gate Array |
| **INCP** | In-network Computing Provider |
| **IoT** | Internet of Things |
| **NFV** | Network Functional Virtualisation |
| **NFVI** | Network Functional Virtualisation Infrastructure |
| **NFVO** | NFV Orchestrator |
| **NS** | Network Service |
| **P4** | Programming Protocol-Independent Packet Processors |
| **PDP** | Programmable Data Plane |
| **PLC** | Programmable Logic Controller |
| **PSA** | Portable Switch Architecture |
| **REST** | Representational State Transfer |
| **RMI** | Remote Method Invocation |
| **RTT** | Round Trip Time |
| **SDN** | Software-Defined Networking |
| **SFC** | Service Function Chaining |
| **SOA** | Service-Oriented Architecture |
| **SOC** | Service-Oriented Computing |
| **TCB** | Trusted Computing Base |
| **TCP** | Transmission Control Protocol |
| **TEE** | Trusted Execution Environment |
| **TNA** | Tofino Native Architecture |
| **VIM** | Virtual Infrastructure Manager |
| **VNF** | Virtual Network Function |

**VNFM** Virtual Network Function

# Definitions

**Trusted Computing Base (TCB):** An entire combination set of protection mechanisms within a computer system, including hardware, firmware, and software, responsible for enforcing a security policy.

**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`

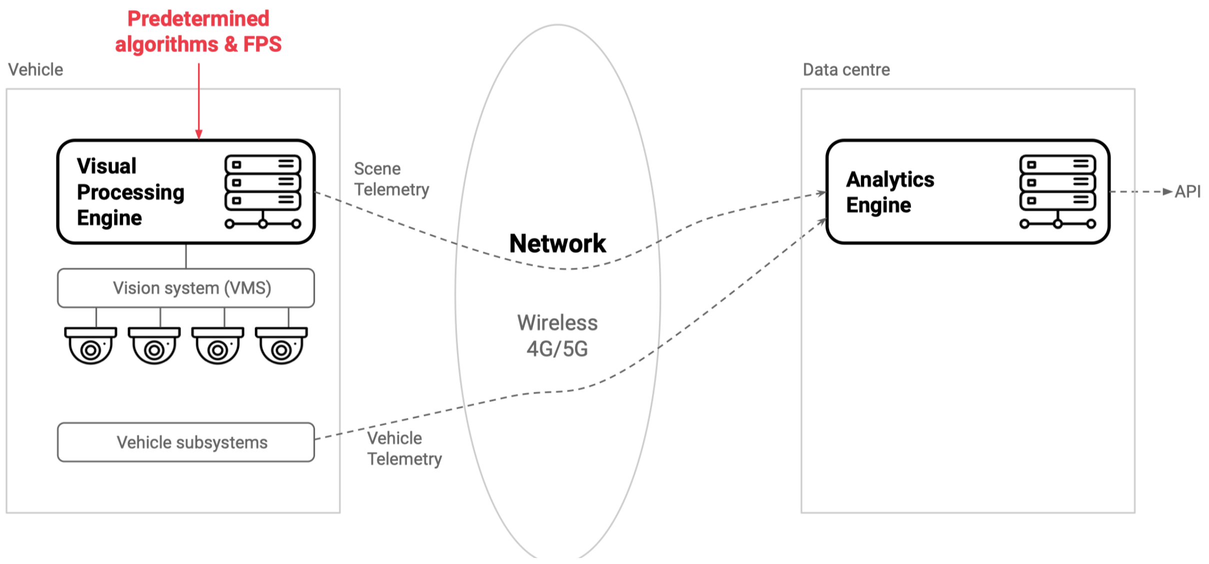


Figure 1: Current set-up: Sensing Feeling Vision Processing

# 1 Introduction

A Piccolo system is a distributed computing environment that runs functions and actors pertaining to a distributed application. Piccolo manages the distribution of those components, provides communication abstractions and APIs, enables efficient resource allocation, availability and scalability support, and other services.

The Piccolo mechanisms and APIs can be implemented for a variety of platforms and programming environments. This document describes some core properties of the architecture identified so far and illustrates it with a specific use case (distributed vision processing).

A Piccolo system consist of a collection of worker platforms (Piccolo Nodes) that can host Piccolo functions. We describe the Piccolo Node concept in Deliverable D2.1. This document is focused on the overall distributed computing aspects, for instance, the communication protocols and interactions between the functions.

The Sensing Feeling Vision Processing use case provides an illustration of the approach. The current implementation architecture is depicted in Figure 1. The figure shows a deployment architecture for a vehicle-based behavioural risk management system (for example in public transport) that consist of a vehicle-based camera array with a visual processing engine and a cloud-based analytics engine.

In this system, the Analytics Engine (AE) is running as a Linux container service in a data centre. The Visual Processing Engine (VPE) is coupled to an array of cameras, the Vision System (VMS), and is connected to the AE over the Internet.

This setup is sub-optimal: Coupling the VPE to the VMS results in relatively inflexible systems: the cameras have to be close (potentially directly connected) to the VPE, and scaling the system (e.g., by

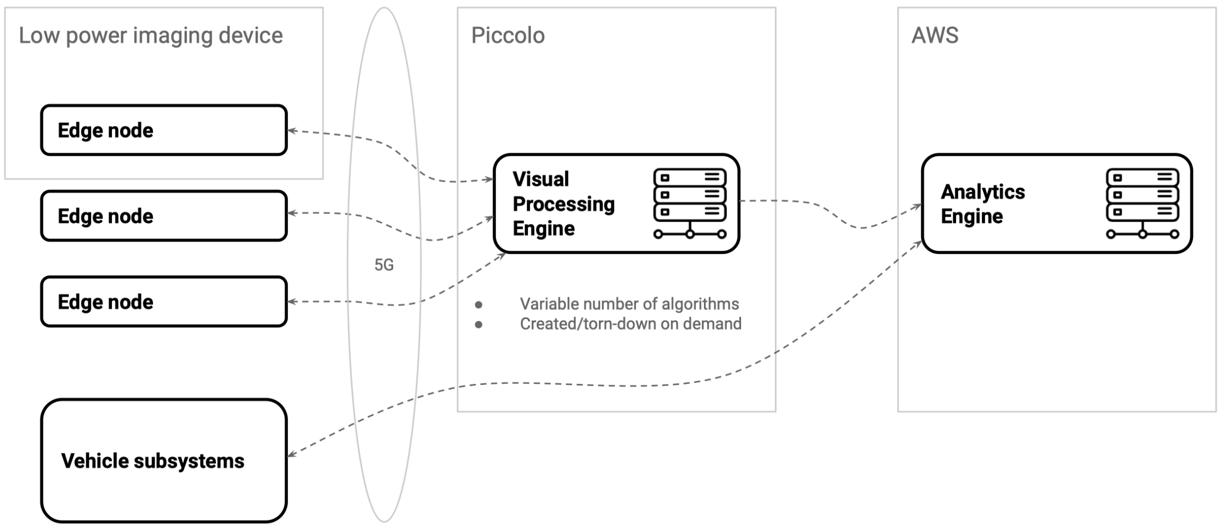


Figure 2: Piccolo Re-Design: Sensing Feeling Vision Processing

adding more cameras) could result in duplicating the whole subsystem (consisting of VPE and VMS). Since VPEs are hosted on full-fledged, potentially hardware-accelerated computers, the system would not scale from a cost perspective.

What is needed is a more flexible system that allows scaling the VMS (the cameras) independent of the VPEs, while still maintaining some control of VPE placement (for example to ensure certain levels of privacy preservation), as depicted in Figure 2. In this setup, the entire functionality of the scenario is encapsulated and implemented as a set of loosely coupled and discoverable components distributed over a network—following the design principles of Service-Oriented Computing (SOC) (cf. Section 2.2.3). In this case, the Visual Processing Engine that processes the actual video data would run on an in-network infrastructure (e.g., an edge-computing platform). It would produce insights from the machine learning classification process (such as the number of people based on certain features/criteria). The analytics engine would (in this particular deployment) still reside in the data centre and consume the vision processing results as well as data from the vehicle subsystem.

The decomposition is attractive for several reasons:

- It addresses privacy and latency sensitivity, as the privacy-critical and comparatively high-volume video data does not have to be transmitted beyond the edge infrastructure.
- It can enable a more cost-efficient deployment, e.g., considering large-scale systems, where a Visual Processing Engine (VPE) can only support a limited number of video inputs. As the number of inputs grows, an operator may want to scale the system. In the decomposed approach, only the (relatively computation-intensive) VPEs have to be duplicated, not the other components, which can make infrastructure usage more efficient. Figure 2 shows a scenario where the VPE resides on an edge computing platform, which could be a trusted platform, so that raw data is never sent to the cloud.

- Similarly, the system could be designed for increased failure resistance at the edge, e.g., by running stand-by VPEs and then switching video streams in the presence of failure (or other events).

The decomposition shown by this example raises several questions to the general distributed computing architecture, for example:

- How do the different data sources, processing functions, and consuming functions communicate, i.e., what are the interaction types (dataflow, data streaming vs. Remote Method Invocation)?
- What are the underlying protocols that implement these interactions?
- How are functions instantiated?
- How is the resource allocation managed?

Deliverable D1.1 [1] describes the different Piccolo use cases and the corresponding opportunities and requirements with respect to applying Piccolo's distributed computing concepts. Deliverable D2.1 [2] describes Piccolo's initial Node Architecture, i.e., conceptual model of execution environments, the corresponding APIs, and conceptual procedures for instantiating and running functions on a particular node (as well as the different specific execution environments that the Piccolo project is working with).

This document is based on these foundations and hence describes the Architectural Invariants, i.e., fundamental principles for later, more specific architecture work to build on, for Distributed Computing in Piccolo. It provides an initial architecture that will be refined and later validated in the Piccolo project.

The rest of this document is structured as follows: section 2 describes key concepts relevant to the Piccolo architecture, and section 3 summarises the corresponding state of the art. In section 4, we describe the initial Piccolo architecture, and in section 5 we lay out future directions and research plans.

# 2 Key Concepts

The Piccolo project aims to provide solutions for use cases with varying requirements over highly heterogeneous hardware. To achieve this, Piccolo will rely on core concepts and relevant solutions and technologies that already exist within the distributed computing space. This includes relying and building on core distributed system principles to answer questions such as *what abstractions to use for designing Piccolo services*, *how to distribute computations in a heterogeneous infrastructure*, *how to provide secure, efficient and optimized services over systems that operate in non-secure environments* and many more.

This section describes several key concepts that form the foundations for such solutions and will provide the necessary background in understanding the Piccolo concepts explained later in this deliverable. Specifically, we describe several relevant specification paradigms, different models of computation, concepts of optimization, orchestration and automation, especially pertaining to service allocation on a distributed infrastructure, and security and privacy of the system-at-large.

## 2.1 Relevant Specification Paradigms

There are a number of current paradigms for developing systems which have some of the features which Piccolo plans to include, however, none covers the full scope and ambition of Piccolo. In each case there are important concepts which can be directly incorporated:

- **Stateless:** The stateless paradigm seeks to avoid holding state which needs to be coordinated. This means there is no need to solve a problem of ensuring consistency of state across the distributed system. (Note, a more precise understanding of this concept is developed in Section 4.2 below.)
- **Serverless:** With serverless, functions which are distributed across the network can be referenced directly by the name of the function and not by the network address of server which is hosting the function. This is a core concept for the Piccolo architecture.
- **Functional Programming:** Functional programming is based on composing and applying functions (maps between values). It has developed from concepts first developed by Haskell Curry and Alonzo Church and implemented in programming languages such as Haskell and ML (and its derivatives such as OCaml). Most modern programming languages provide the necessary facilities for functional programming. In pure functional programming, functions have a precise and restrictive template where functions cannot hold state between calls, so that the response is entirely determined by the input parameters and the function specification (and no other effect on the system outside the function call). Functional programming has a number of features which are important to Piccolo including:
  - functions are ’first class citizens’ and can be passed as parameters to other functions;

   - a pure function can be rigorously tested as every call with the same input parameters must have the same response;

   - some functional programming languages require the parameter type of the response to be fully specified making it possible for the compiler to do full type checking at compile time;

   - the functional paradigm makes system evolution, refactoring, and CI/CD development much easier as module testing is simple and reliable.

- **Object Orientation:** Object orientation has been a mainstay in programming since the widespread adoption of C++ in the 1980s (it was originally developed in Simula 67 and Stroustrup [3] directly imported the concepts as a superset of C to create C++). Object orientation binds state to functions which can access and change the state and therefore enables the encapsulation of state. However, it is normally implemented using a synchronous calling model which is not well suited to distributed systems. The actor model (which is described in more detail in the next section) is a form of object orientation and has similar encapsulation of state but is based on an asynchronous calling model (ironically, Simula 67 included direct support for what was essentially an actor model!). Object orientation has had other difficulties, notably in defining abstractions of objects where the requirements of code reuse at coding time can conflict with the requirements for polymorphism of method calls (for example, the Julia programming language has emerged in part as a result of this conflict and gives explicit priority to polymorphism).

- **Systems, Processes, and Finite State Machines:** While most programming paradigms have grown up with a model of processing a task in non-realtime environments, many realtime systems are designed using a continuous flow paradigm which is common to the design of realtime physical systems (e.g., SysML [4]), process design (e.g., BPMN [5]), and finite state machines. While stream processing in some modern languages (e.g., Scala) gives some bridge between the two, Piccolo architecture may need to draw directly from these continuous flow paradigms.

- **Network Protocols and Sessions:** Network protocols have been developed in a different environment from either programming languages or continuous flow processes, although they have many features in common with both. Network protocols run state machines (which may be either explicitly or implicitly defined) and understand that the network is unreliable. Coordination of state across the network is a fundamental issue and there is a strong preference for either stateless protocols or protocols which are tolerant of inconsistent state. Where state is coordinated across the network, it is held in the context of a session which has a well defined lifecycle so state does not continue to be held when its context lost. These are important concepts for Piccolo.

- $\Pi$ **Calculus [6]:** This forms a formal mathematical framework for distributed systems and can be taken as the equivalent for distributed asynchronous systems that lambda calculus is for central synchronous systems. While the formal framework is important for Piccolo in its own right, there are also a number of important insights which can be taken directly into the Piccolo architecture, notably: 1) functions are processes and 2) all state is held by a process and only accessible through process calls.

## 2.2 Models for Computations

Every application that is deployed on an infrastructure can be viewed as a functional black-box that (i) takes external input, (ii) performs certain computations (in either stateful/stateless fashion) and (iii) generates output. In traditional terms, the application black-box (step ii) can be represented as different models of computation—each differing on how functional components are executed and dependencies that exist between them. In this section, we present a few key models of computation for modelling and executing Piccolo services.

### 2.2.1 Stream Processing & Dataflow Networks

Stream Processing is a software paradigm based on the idea of disconnecting computational actors into stages that can execute concurrently. Stream processing is essentially a compromise, driven by a data-centric model that works very well on self-contained data where all necessary data for processing is local, e.g., digital signal processing (such as image, video). However, it is less suitable for general purpose processing with more randomized data access (such as databases). By sacrificing some flexibility in the model, the implications allow easier, faster and more efficient processing of huge datasets. Further, there are different styles of processing like batch processing or a continuous processing of a stream which have different performance and real-time characteristics. Stream Processing is usually done in an acyclic fashion, i.e., the pipeline of computational actors can be described as a directed acyclic graph. A relevant application is Apache Flink (see Section 3.2.1) which leverages continuous stream processing. Within Piccolo's context, the Vision Processing use case of Sensing Feeling (described in Section 1) can leverage stream processing to handle a large number of video inputs at the VPE, which is deployed in-network.

A more general concept is the dataflow network which is a network of concurrently executing processes or automata that can communicate by sending data (messages) over channels. Abstractions of such a dataflow network can serve as a system design model for distributed computing. The topology graph of such a network is directed and can also be cyclic. Notable applications are found in Erlang (see Section 3.5.1) and Java/Akka which leverage dataflow networks using the concept of actors. Another relevant application can be found in IEC 61499 (see Section 3.3.1) which provides a system level design language to specify dataflow networks consisting of function blocks for industrial process measurement and control systems. It is possible to reason over dataflow networks using, e.g., the pi-calculus (see Section 2.1).

### 2.2.2 Actor Model

The actor model of computation [7, 8] models all computations using actors. Actors are computational entities that only communicate using messaging and maintain local, mutable state that is not accessible by other actors. Together with the program code ("script") this state defines the actor's behaviour when processing an incoming message [8]. While processing a message, the actor may create additional actors, asynchronously send messages to other actors, and modify its local state, which influences the processing of the next message [8, 9]. The scope of the actors varies. While the theoretical model assumes that everything including the messages and the computational primitives are modeled using actors and that the primitive actors might be implemented in hardware [7], practical implementations such as Erlang or Akka often use actors as a grouping and concurrency-control mechanism for higher layer entities [10].

Compared to the dataflow model, the actor model allows for the creation of additional actors and dynamic changes to the network topology [8]. It is commonly known that, as the state is local to the actors, which are only activated by the arrival of a single message at a time [8], the actor model prevents concurrency issues that could arise from shared state. The locality of the state allows for clear reasoning about its validity in scenarios such as horizontal scaling of application components or network partitions. The actor model can be applied to both actors running on the same device as well as actors running on multiple networked devices [8], and the exclusive use of messaging provides a natural and location-independent interaction method between two application components in a distributed system. These properties make the actor model a possible abstraction for computations in Piccolo.

### 2.2.3 Service-oriented Computing

SOC or Service-Oriented Architecture (SOA) describes a paradigm in which a set of loosely coupled distributed compute components (e.g., services) can be discovered and invoked in the network (e.g., [11]). The paradigm covers the following main aspects [12]:

- Applications are developed as a *set of loosely coupled components*—called services—in contrast to a monolithic program. Each service is in itself *self-contained*.
- Each service provides a *well-defined interface description* providing contextual information as well as public information such as the supported interactions of the service.
- Services are in nature *distributed in the network* which can be invoked by end-customers or other services using their provided interfaces.
- Interfaces encapsulate the functionality of each service and ensure *discoverability* of the service within the network.

The characteristics of the service-oriented paradigm, especially the fact of encapsulated functionality as loosely coupled and discoverable services, is a promising approach to handle distributed applications that change at runtime.

In the past decades, the SOC principles have been adopted in several areas of applications including mobile operating systems (e.g., Android OS [13]), enterprise software and in Web Services [11]. While SOA is an architectural style, actual frameworks and communication middleware implemented the service-oriented design principles. Popular examples include the standardized architectures such as the Open Services Gateway initiative (OSGi) [14], the Data Distribution Service (DDS) [15] or the OPC Unified Architecture [16] for robotics, automotive or industrial applications respectively.

### 2.2.4 Microservice Architecture

The microservices architecture is a development of Service Oriented Architecture (SOA) and is widely adopted in the design of systems, especially systems which have some inherent distribution. The concept and practice has evolved over time and as a result, there is no one simple or clear definition and different aspects can be emphasised in different circumstances [17].

Broadly, microservices architecture involves designing a system as a set of smaller component services where each component carries out one well defined task. The execution of each component task is offered as a service which can be called across a network and each task execution is a discrete response to each call.

Importantly, the scope and design of the component is more oriented to what it intrinsically does rather than deriving solely from the specification of the wider system. This means that it is more likely the microservice can be reused without modification either in later updates of the wider system or in other systems. This marks a significantly shift in the approach of *code reuse*. Previously, the main focus of code reuse was only at coding and compile time and focused on features like design patterns, templates, and class inheritance. Code reuse in microservices architecture focuses on reuse at runtime.

Each component offers a service interface - a microservice - and other components can call this service at runtime. This allows a complex end to end systems to be composed from a set of components by designing each component to call the services of other components as needed.

This offers a number of advantages to the development, integration, and deployment lifecycle. It is widely adopted for systems which use the continuous integration/continuous deployment (CI/CD) development methodology. Each microservice component can be separately developed, improved, and updated with associated testing and validation without causing significant integration changes to the rest of the system.

Another advantage of microservices architecture is that it is much easier to decouple functional logic from allocation of resources to achieve a required performance. Each microservice component can be separately allocated resources using standard hosting techniques which can achieve automatic scaling of resources in response to changing demand for the microservice.

The success and popularity of the microservice architecture has led to the widespread adoption of a number of common standards and hosting environments. This is so much the case that these are often used as an informal definition of microservices architecture. The common standards and hosting environments include the following:

- The use of HTTP and HTTPS REST interface standards for calling the service and receiving responses. This places full responsibility for maintaining consistency of state on the application itself. A specific REST interface of a microservice is often specified and published in OpenAPI using the Swagger support tool.
- The implementation of a micro service as a container and hosted in a container environment.
- The use of notably Kubernetes to manage the dynamic scaling of container instances to match compute resources to the demand for the micro service
- The use of a message bus, notably Kafka, to route call and response messages between micro services.

While these are all widespread, for some applications they are they are overly 'heavyweight' and the resources required to support the environment can be greater than that required by the application. For example, the Erlang environment (discussed in Piccolo deliverable D2.1) fully supports all the fundamental features of microservices architecture without using any of the above standards or hosting features.

Microservices architecture has many of the features required for the Piccolo architecture and if taken in a very broad sense, Piccolo architecture is a form of microservices architecture. However, when compared with the common realisations of microservices architecture, there are several gaps, including the following.

- The container can be quite a large unit for some components
- The focus of performance and reliability with resource assignment is restricted to each component.
- The scaling is linear and essentially neutral to location and parameters such as latency and other network costs.
- Scaling is normally achieved using dynamic load sharing in the dataplane and can become a bottleneck.
- The REST call format defines the network location before the function which restricts mobility mechanisms
- The indirection between microservices using a message bus (eg Kafka) in the dataplane creates

its own complexity and possible performance impact and is also not well suited to data streams such as video.

The impact of these and how Piccolo can overcome them is developed in Section 4.

## 2.3 Orchestration, Optimization and Automation

Within this section, we will cover three key concepts—Optimization, Automation and Cost of Change—and how they relate each other and to Piccolo. Orchestration is a well-investigated topic, and the definition of Orchestration varies in different fields.

As an example, within the ETSI NFV specification, orchestration is dealt with by an NFVO; this function is defined as the "functional block that manages the Network Service (NS) lifecycle and coordinates the management of NS lifecycle, VNF lifecycle (supported by the VNFM) and NFVI resources (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity"[18].

Within the work done by CNCF in the Orchestration space, there is no explicit definition of what is included in Orchestration. Instead, the CNCF landscape has the notion of "Orchestration and Management", which includes many sub-functions such as "Scheduling and Orchestration", "Remote Procedure Call" and "Service Mesh"[19]. The CNCF also has a "serverless landscape", which is lacking this explicit Orchestration layer[20].

While Piccolo is not necessarily focused on only NFV based applications, there is an overlap between the ambition of Piccolo orchestration and what is designed by ETSI, as well as with some of the technologies listed by the CNCF in their landscape. The expectation is that Piccolo will require additional functionality beyond ETSI / CNCF specifications—especially as existing solutions focused on virtual machines, or containers (although these may still be used within the underlying execution environment).

### 2.3.1 Optimization

While optimization and orchestration are distinct functions, an orchestration layer can include an optimization function within it. Optimization is aimed at finding the best solution, within a set of solutions often based on the minimisation or maximisation of some objective or goal. Within Piccolo, there are several goals this optimization function within the orchestration might be attempting to minimise, some examples of which are listed below:

1. Compute or network resource utilisation.
2. Instantiation time of the function.

3. Latency from the function to the relevant consumer.

Some use cases will optimise towards all, a subset or a singular objective, this will be driven by some cost-reducing exercise and policies implemented for each use case. Additionally, within this optimization process, there will be acceptable trade-offs, for example, a use case may sacrifice latency for reduced compute utilisation, again driven by policies defined for the use case. Across these objectives, common themes can be distilled, which impact these objectives to different magnitudes.

- Our relation to other functions within the compute fabric.
- The location of instantiation within the network concerning these other functions.
- The location of instantiations of those who are consuming the functions, either by sending inputs, or receiving outputs.
- What hardware resources the function requires.
- Policy requirements defined by the tenants, including security, and resource partitioning or sharing with other functions.

**Joint optimization of computing and network resources**

Because Piccolo is not conceived as a Virtual Network Function (NFV) overlay—but rather as a lightweight distributed computing system with visibility into the current network and compute performance as well as into the resource utilization—it can provide a deep integration of networking and computing. The Piccolo elements can perform self-organised optimization without the overhead and scalability issues of traditional fine-grained control and orchestration systems, thus reducing cost and complexity for operators. The optimization and distribution needs to be cognizant that the compute and networking resources are highly heterogeneous. It can range from ARM-based chips at the edge to multiple racks of x86 servers deployed in regional data centres. It can span existing cloud platforms, network infrastructure, user devices such as mobile phones and home gateways.

Considerations should be given whether specific hardware requirements might speed up the execution of a function at the edge. For example, should the optimiser prioritise ensuring that a function executes for the shortest duration, potentially at the cost of more expensive or limited hardware resources? The optimization function becomes more complex if we consider that the function can impose hardware constraints for operation, e.g., a VPE unit (described in Section 1) can only operate on edge resources with an associated GPU.

**Optimization of function instantiation**

Considering instantiating a function on edge infrastructure as an example, thought must be given for the type of compute the function is instantiated on as well as where in the network to instantiate it. This point is especially important as wherein the network a function is instantiated can have a significant impact on not only how it performs, but how efficiently resources are utilised in both the network and compute layer.

[21] provides an example for ICNs, namely, minimizing routing costs by jointly optimizing caching and routing decisions over an arbitrary network topology. This work considers both source routing and hop-by-hop routing settings, and the corresponding algorithms reduce routing costs by several orders of magnitude compared to prior art, including algorithms optimizing caching under fixed routing.

**Optimization of latency from/to function**

Another aspect of function instantiation is how long this process will take. We can consider that it will take time to download the required code, and possibly execution environments, to a compute element in the network. For example, would it be "cheaper" from an optimization standpoint to re-use an existing compute element that might incur additional network, compute or "Quality of Experience" cost to the function. Temporal elements are another aspect with optimization. A solution that is implemented now might be optimal now, but this optimality reduces over the lifetime of the solution. Conversely, you could apply a sub-optimal solution now, with the assumption it will become more optimal in future. This is a field often dealt with as "Q-Learning" within the machine learning community.

### 2.3.2 Automation

One of the key characteristics of Piccolo functions is their event-driven nature, which requires rapid invocation of the function in response to an event, such as an API call, especially with the expectations that transactions will be short lived. Expecting a human-driven process to respond at these time scales is undesirable, making automation crucial. Existing technologies, such as micro-service designed applications using containerisation, could be manually instantiated at a small scale, but at a larger scale, automation is a necessity for instantiation, scaling and management—this gap is often filled by software such as Kubernetes.

As mentioned, instantiation of Piccolo functions, like other FaaS technologies, is event driven (e.g., in response to a request hitting an endpoint or some change in the data state). As a result, even on a small scale, automation is a necessity to minimise instantiation times as well as managing the chain of events required for a successful instantiation (e.g. image onboarding to compute node).

This automation is also required to scale the instantiation of functions where millions of events are being registered. The project also has additional considerations as a result of the In-Network Compute paradigm, which means that the functions may benefit from additional optimization, such as with function placement versus other, dependent functions, as part of this automated process. The project should also consider that it is not just the instantiation of a function that must be automated, but also (due to their highly ephemeral nature) nearly every element of a function's lifecycle, including but not limited to onboarding, scaling, monitoring, teardown, licensing and package management.

### 2.3.3 Cost of Change

At all stages of orchestration, especially within the optimization, considerations must be given to the cost of change. What is meant by this, is when taking an action, how much will it cost us not only in the immediate term but over the lifetime of the function. The definition of cost is a difficult one to quantify, as it will depend on the inputs required for any optimization function, and the goals it is trying to minimise or maximise.

It's also important to consider the cost of not changing—without this guiding principle, the platform would risk not instantiating a function to avoid the overhead of compute / network usage. For example the cost of not gaining potential revenue from the execution of the function. Below are characterisation of two key changes that occur within the scope of Piccolo:

**The invocation of a new function, or a series of functions**

The instantiation of a function brings associated costs, even in the case where prerequisites for the execution of the function already exist on a specified compute node (there may be an overhead for initialising the underlying execution environment for example). Where the required prerequisites do not exist, especially if missing the required code and dependencies, then additional start-up cost is incurred.

**The destruction of a function, or series of functions**

When a function has finished executing, consideration must be given whether the function should be kept "warm", and then recycled in future or whether to discard it. Keeping it in a "warm" state incurs costs with CPU cycles and RAM usage but avoids any potential overheads of instantiating execution environments (although in some cases, this might be negligible). Piccolo must also assess the long-term storage of the function, specifically, the code and dependencies which must be stored. Any locally stored state will be lost as well; however, the session state will likely be stored externally, with the only local state being what is required for the completion of the current transaction.

## 2.4 Decentralization and Resource Allocation

Decentralization refers to the outsourcing of computation tasks to multiple entities that possess computing and storage capabilities and can communicate via a network. The emerging Internet of Things (IoT) and the 5G architecture desire applications that respond in sub-millisecond latencies [22]. Even though the centralization of compute behind data centres has served well the purpose of the today's internet and was also in agreement with the demand of economies of scale, it is surely not well suited for next-generation applications. Such low-latency applications cannot tolerate the response times of centralised computations taken care of by far-away data-centres. The proposition of Piccolo is based on the decentralisation of the cloud into smaller scale computing devices (ranging from mini-data centres to single-board computers), that contribute their computation power and storage space to requested tasks. This trend results in minimisation of data transmission overheads (between the edge and the cloud infrastructure) and improves the performance of computing in Cloud platforms by diminishing the volumes of data that needs to be processed and stored. Furthermore, decentralization, by design, also enables privacy preserving applications since the requirement for offloading sensitive IoT/user data to a cloud datacentre is no longer required. Several recent works (such as [23]) have showed possibilities to enable privacy-sensitive applications at-large using techniques such as selective denaturing – which are fundamentally dependent on the availability of a widespread decentralized compute infrastructure.

Decentralization also spawns the problem of effective resource allocation for executing services. Resource allocation plays a vital role in application provisioning and affects the overall QoS of a system. Allocating resources over a geo-distributed (i.e. nodes distributed over different geographic areas) set of heterogeneous edge nodes like in Piccolo differs from typical resource provisioning as the allocation scheme needs to take into consideration the impact of network conditions (i.e. latency between end users and Piccolo Nodes) and the load of each node. Moreover, for mobile users, like in the case of smart vehicles, the latency to the allocated resources changes as soon as the user hands over to another base-station. Furthermore, some of the resources within the network, especially in a highly dynamic environment, may be available only within temporal and spatial capacity – but can provide an excellent QoS if selected. The optimization principles behind resource allocation and function placement in a decentralized infrastructure was explored in Section 2.3. However, it must be noted that even optimal resource assignments get outdated over time and approaches such as static provisioning lead to idle resources, thus presenting several unique service placement challenges that Piccolo will tackle throughout project scope.

## 2.5 Security, Privacy, Isolation, Attestation

Security and privacy concerns in computer systems have increased in significance with the ubiquity of connected devices. Private data is stored and processed not only in multi-tenant cloud providers but also at the network's edge, where privacy preservation is more demanding due to the distributed structure of such environments. In this section, we introduce the fundamental concepts of security, privacy, isolation and attestation along with mechanisms and policies that ensure the protection of code and data.

### 2.5.1 Security and Privacy

In principle, security involves the protection of confidential information from being revealed to malicious adversaries and incorporates mechanisms to protect data through its different states:

1. **Data in transit:** refers to data being transmitted between two points over a medium (network).
2. **Data at rest:** refers to data that is not being accessed and is stored on a physical or logical

medium.

3. **Data at use:** refers to data being opened by one or more applications for its processing or consumption by users.

**Data in transit**
Regardless of whether information is traveling across a public or private network, there is a need to ensure the secure delivery of digital evidence to maintain its authenticity and integrity. Protecting data in transit is a combination of network security and encryption techniques in order to overcome threats such as eavesdropping or man-in-the-middle (MITM) attacks. Encryption of data in motion should be mandatory for any network traffic that requires authentication or high-level privilege. However, there are mainly two aspects of encryption:

**Encryption in-transit:** This combines encryption mechanisms at the transport layer and usually follows encryption standards like SSL/TLS (Secure Socket Layer/Transport Layer Security) certificates while also incorporating techniques like secure connections through VPN tunnels, IPsec or SSH tunneling.

**End-to-end encryption:** Encryption in-transit is useful, but it does not guarantee that the data will be encrypted at its starting point and will not be decrypted until it is in use. End-to-End Encryption (E2EE) is a communication paradigm where data is encrypted before it is transmitted and will remain encrypted until received by the end-party.

**Data at rest**
Securing data at rest involves different types of technologies and authorization mechanisms ranging from file-level encryption and Database Encryption to digital rights management (role-based access and authorization) and data leak prevention using policy packs driven by regulatory compliance.

**Data at Use**
Private data is generally revealed to the party performing the computation on it. This poses an issue, particularly when outsourcing storage and computation either in cloud environments or edge infrastructures with the latter being a more challenging case as the heterogeneity of devices contributing to computation tasks is not always controlled by the same entity.

Currently, there are typically two types of techniques to achieve secure outsourced computation:

- theoretical cryptography solutions, and
- system security solutions.

In terms of cryptography, homomorphic encryption [24] is considered as a promising solution for outsourced computation. It allows third parties to perform computation on the encrypted data without exposing the content of the plaintext. However, the computation overhead is still colossal, which is not fit for practical usage [25].

On the other hand, system security solutions have grown rapidly the last few years with the introduction of Trusted Execution Environments (TEE). Solutions like Intel Software Guard Extensions (SGX) and ARM TrustZone have been developed to achieve secure computation by allowing user-level or operating system code to define private regions of memory, commonly referred to as enclaves. Data inside enclaves are shielded and cannot be either read or modified by any process outside the enclave itself. In general, the TEE achieves a performance level equivalent to the plaintext computation overhead. However, TEEs are prone to side-channel attacks and the information inside the enclave can be leaked to adversaries [26].

### 2.5.2 Isolation

At the core of Piccolo is the sharing of computing and communication resources among co-located, multi-tenant applications. Underlying resources of Piccolo nodes are shared between multiple entities for optimizing resource utilization and energy efficiency. Isolation of resources is very critical when systems contain applications with mixed criteria in terms of privacy or prioritization potentially sourced from different vendors to handle multiple complex operations. Piccolo will leverage from different virtualization techniques, from Virtual Machine and Containers to Unikernels, in order to provide a secure, flexible and scalable architecture for diverse environments.

**Virtualization**
In principle, virtualization technology allows the sharing of the same physical resources among many users. A software layer is used to enable the sharing of hardware between the different users. In virtualization, a hypervisor provides a layer between a guest OS and the physical hardware. The hypervisor is responsible for communicating and sharing the hardware by scheduling the CPU time for each guest and simultaneously allocating virtual memory from the physical memory to each VM. Strong isolation is achieved through the hypervisor as it is the only entity that can communicate directly with the hardware being at ring level 0 (kernel space).

**Containers**
Operating System-level virtualization is a technique where the kernel of an operating system allows for multiple isolated user-space instances. These instances, usually called containers (LXC [27], Docker [28]), Zones (Solaris [29]) or jails(FreeBSD jails [30]), run on top an existing host operating systems and provide a set of libraries that applications interact with, creating the illusion that they are running on a dedicated machine. OS-level virtualization usually imposes little to no overhead as the programs in virtual partitions use the operating systemâs default system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine.

The most famous container technology, Docker, is an open source application container platform designed for Linux and more recently Windows (through a Linux VM that runs using Hyper-V, the windows native hypervisor) and MacOS (through the Mac OS X hypervisor framework). Its simple usage, low-overhead and great scalability makes it ideal for the micro-service architecture where each micro-service is considered as a set of co-operating containers.

In terms of isolation, containers leverage from Linux control groups and namespaces in order to guarantee safety, security and confinement. Each aspect of a container runs in a separate namespace and its access is limited to that namespace. While this spatial isolation limits the impact of malware to a single container, the large amount of shared resources (e.g. main memory, I/O controllers, system bus) imply that single points of failure exist and software running to one container can impact software running in another container due to interference. Therefore, OS-level virtualization technologies tend to have a weaker trust boundary between them and the host and thus are more prone to software attacks and unauthorized access.

**Unikernels**
Unikernels [31] are an alternative concept in which an application is decomposed to determine the required system modules (kernel dependencies). Only the necessary components are compiled along with the application binary into a bootable and immutable image. Some of the characteristics of Unikernels are a small size (only a few MB), low overhead, minimal attack surface and rapid boot process. Those features make Unikernels attractive for edge/fog computing environment where functions roam within the network.

### 2.5.3 Attestation

In principle, attestation is a mechanism for software to prove its identity and is usually a feature of Trusted Platform Modules (TPMs) [1] and Trusted Execution Environments (TEEs) [2]. The most popular paradigm of attestation is remote attestation which is a method by which a host (client) authenticates its hardware and software configuration to a remote host (server). The purpose of this operation is to enable a remote system, usually referred as challenger, to determine the level of trust in the platform's integrity of another system which is referred as attester. Remote attestation may be used to address a number of trust problems including guaranteed invocation of software/applications, delivery of premium content to trusted clients and alleviating mutual suspicion between clients.

There are five principles [32] that are crucial for all attestation architectures. However, not all principles can be satisfied by real systems (e.g. RIoT [33], Intel SGX [34] and MIT Sanctum [35]) and thus vendors' attestation mechanisms may provide better implementation of some features than others. The basic five principles are:

**Freshness:** assertions about the target should report recent state, code and data.

**Sufficient Information:** attestation mechanisms should provide sufficient information to allow challengers and trusted proxies to make correct decisions.

**Trustworthiness:** measurement and reporting should be done by a trustworthy Trusted Computing Base (TCB). State, data and execution context cannot be tampered and accessed by entities outside of the trust boundaries.

[1]TPM: tamper resistant piece of cryptographic hardware built onto the system board.
[2]TEE: An area on the chipset that works like a TPM, but is not physically isolated from the rest of the chip.

**Constrained Disclosure:** The attester can decide what information can be sent to challengers without revealing confidential and private data.

**Transparent Semantics:** The semantic content of attestation should be presented in a uniform and logical form to accurately determine the identity of the target so that a challenger can collect attestations about it.

Parties that offload their software to remote platforms owned and maintained by untrusted entities need to have guarantees in terms of confidentiality and integrity. Piccolo could leverage from attestation mechanisms (depending on the hardware platform) in order to assure that critical operations are executed in tamper-resistant environments hosted by trusted and certified hardware.

# 3 State of the Art In-Network Computing

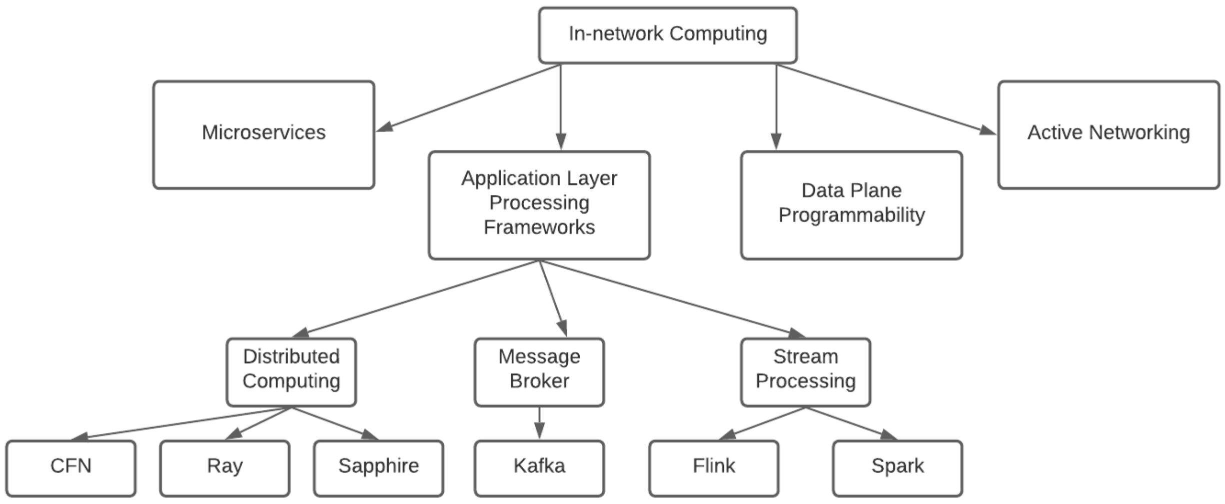


Figure 3: Taxonomy of Different Approaches for In-network Computing

As in-network computing is realized in distributed systems, it aims to provide integration between computing and networking to achieve better performance in terms of latency, throughput, and privacy. There exist different approaches that implement in-network computing [36]. Some implementations consider offloading part of the system's functionality to network devices like P4 switches or edge computing platforms. Other systems executed in a distributed environment, such as microservices and some application layer processing frameworks, distribute the application's functions to multiple nodes across the distributed system. "Active Network," [37] as an approach in this context, allows computation to be implemented using network switches while allowing the network elements' real-time programmability. "Active Messages" [38] also combined computing and communication by inserting a handler's address in a message header. Then this handler will be executed directly with the message arrival while having the message content as an argument.

Different programmable network devices as FPGAs and ASICs along with a programming language as P4 have been introduced to perform part of the computation logic in the network. Further details are mentioned later in the chapter.

Multiple systems adopted the distributed computing pattern. It is presented in microservices architecture and in many application layer processing frameworks. As described in Section 2.2.4, microservices are about dividing complex applications into multiple loosely-coupled small services. This architecture has been widely adopted due to its benefits in handling complex applications as it proved its advantages in maintenance, scaling, and fault tolerance.

Application layer processing frameworks implemented in a distributed environment are another category of distributed systems. There are general-purpose processing systems like Ray [39] and Sapphire [40]. These systems are designed to simplify the programming of complex applications that span heterogeneous environments and require complex distributed tasks.

Also, there is the set of systems specialized in stream processing, as explained in Section 2.2.1. Apache Flink is one framework that is used for processing stateful computations over unbounded data streams. Its competitor, Apache Spark is another distributed data processing engine that is used in a wide range of applications. These frameworks are designed to support large amounts of data. Kafka as a message broker is also another example of a distributed system. It is used with other frameworks as Flink and Spark decoupling the producers from data consumers and providing fault tolerance.

An important system to mention in this scope is Compute First Networking (CFN). It is implemented on top of ICN networking architecture providing solutions for the challenges in distributed systems while leveraging the ICN properties and more importantly providing joint optimization of computing and networking resources which is an essential property we are looking to have in our PICCOLO system.

Microservices and application layer processing frameworks rely on different technologies to execute their functions and manage the overall system. These technologies such as containers, orchestrators as Kubernetes, and actors are further discussed in this chapter as a part of different distributed systems.

The rest of this chapter will talk about these technologies, about some of the previously mentioned systems while including more details about how they are implemented, e.g., their architecture and their network stack. In addition, we will see how they deal with challenges such as fault tolerance, orchestration, management, and scalability. As we are considering these frameworks, they still represent a subset in the state of the art, but we will later highlight how these different systems and technologies are relevant to our Piccolo system.

## 3.1 Data Plane Programmability

The Piccolo infrastructure architecture is expected to support different types of compute and networking devices. In recent years, research activities have worked towards flexible, programmable network devices to meed demands of data center operators and data center switches. As a result, Programmable Data Plane (PDP) have been introduced to recast the network as a deeply programmable platform.

The introduction of PDP enables the implementation of user-defined forwarding functions in network elements using higher-level modeling languages, while also permitting runtime configuration of these devices like in Software-Defined Networking (SDN). These devices usually include programmable parsers with several reconfigurable match-action tables implemented using an FPGA or as an Application-Specific Integrated Circuit (ASIC). Figure 4 illustrates a pipeline of match-action tables implemented in such programmable data planes.

By operating PDPs, a network operator is able to define certain action which will be performed in a PDP device when a packet matches a certain rule. Several of those rules and actions can be programmed by the device operator during deploy time. We refer to this capability as ’horizontal programmability’ of a network device (since across the pipeline). Furthermore, PDPs also implement

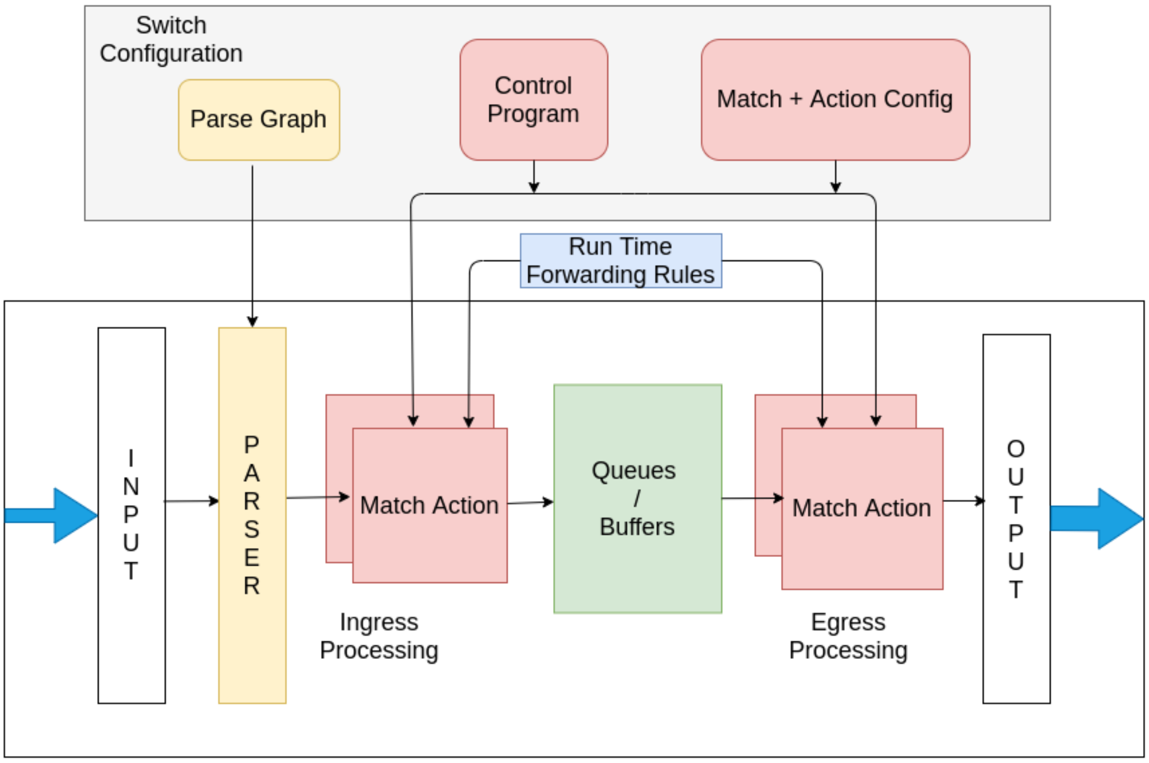


Figure 4: Forwarding Pipeline in a Programmable Data Plane

the SDN paradigm and its vertical programmability (logically centralized control plane configuring the underlying data plane). During runtime, it allows to re-configure the behavior of the switch by modifying match/action rules.

There are two main components of the hardware programmable data planes—an architecture determining the arrangement of the match-action tables and parsers and a modelling language to express the forwarding function to be implemented on top of this architecture (e.g., Programming Protocol-Independent Packet Processors (P4) [41]). While there are several architectures available following the match-action table paradigm (the generic Portable Switch Architecture (PSA), Tofino Native Architecture (TNA) for Intel Tofino ASICs, SUMESwitch for NetFPGAs), P4 is slowly moving towards becoming the de-facto modelling language for programmable data planes. Together, they allow the implementation of user-defined forwarding functions into the network elements. These functions may include limited arithmetic, and thus, implement computational functions directly into the network element.

### 3.1.1 Hardware Architectures for PDP Languages

An architecture is more of a contract between the program and the hardware capabilities. The architecture describes the underlying capabilities of the hardware and exposes interfaces to hardware blocks. The hardware vendor typically provides the architecture, and hence, this is vendor dependent.

From the several architectures available on different hardware platforms, we discuss the PSA architecture from the P4.org Architecture Working Group, as this looks to be on its way towards become the standardized architecture for the P4 language [42].

PSA is an architecture that dictates the bare minimum capabilities that are to be supported by a network device, which employs the $P4_{16}$ (current version of P4) model. This enables the P4 program to be portable across all types of hardware. P4 programmers can use this architecture to write ASIC vendor agnostic P4 code. This is because the ASIC vendors are bound to support the PSA architecture. The standard says as an analogy, "the PSA is to the $P4_{16}$ language as the C standard library is to the C programming language". The PSA defines its own set of libraries or hardware blocks that are to be supported by the hardware vendor. This could be for instance an extern such as counters, meters, registers and so on. In addition to that, the architecture also mandates the ASIC to support itself with the different types of ports such as the CPU Port, Recirculation Port etc.

### 3.1.2 Network Modeling Languages

P4 can be seen as a modeling 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 to reconfigurable architectures like FPGAs.

A P4 program starts by defining the different header fields of the supported communication protocols followed by a programmable parser (implemented as a state machine) which extracts these header fields from an incoming packet. It subsequently defines control blocks which use the extracted packet features to match and execute user-defined actions ranging from dropping the packet to forwarding it on certain ports. The P4 Runtime [43] makes it possible for a control plane to populate the look-up table entries.

P4 also provides constructs to express minimal computing tasks in the forwarding functions enabling offloading of computations into the network. 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.

### 3.1.3 PDP: Networking Applications

As presented in the previous sections, P4 brings programmability to the networking domain and can be used to realize in-network computations. Hauser et al. [44] present a comprehensive survey on programmable data planes using P4 including fundamentals, advances, and applied research.

The flexibility provided by PDPs comes in handy to implement networking applications which so far were implemented in middleboxes, e.g., firewalls, load balancers. Silkroad, a stateful Layer 4 load balancer, manages to distribute the load across a bank of servers at line rate using commodity P4 switches [45].

Service Function Chaining (SFC) [46] and P4Guard [47] are examples of pushing logic from middle-boxes into the network elements.

With SDN, the intelligence required to manage networks was (logically) centralized into commodity servers. While the centralization simplifies implementation of network management logic, it also introduces additional latency in the decision making. While in most cases this delay is non-critical, in some cases like link failures this delay may prove costly. With P4 certain decision making can be handed over to the data plane, while still retaining the SDN paradigm. For instance, SDN controller can program an ordered set of output ports for a flow in a P4 switch, which then outputs the corresponding packets on the first available port [48].

Gathering the network state is a critical step towards efficient management of the networks. Current state-of-the-art mechanisms like sFlow, NetFlow etc., used to gather network telemetry, sample the traffic at the data plane or summarize the statistics based on the 5-tuple definition of a flow. With PDPs, in-network telemetry can be achieved by gathering information at the packet level by inserting additional headers in the packets while traversing the networking and removing them at the egress, without noticeable effects on the latency or throughput. With the gathered information, a coherent network telemetry can be generated at a packet level [49]. In the context of Piccolo, the flexibility of PDPs to adapt the behavior of those devices to application demands (horizontally and vertically) offers new opportunities to assist management and orchestration of compute functions within the network.

## 3.2 Application Layer Frameworks

This section describes some of the application layer processing frameworks that are executed in a distributed environment. These systems present a subset of the multiple interaction paradigms that Piccolo aims to support while providing some good mechanisms and concepts in how they deal with distributed environment challenges. These challenges may vary from management and orchestration to scalability, fault tolerance, etc.

In this scope, we can mention Sapphire [40] which serves in decreasing the complexity of distributed applications by handling the deployment of these applications across a heterogeneous environment.

Ray, another distributed framework that targets AI applications implemented in a distributed environment. Ray supports stateful and stateless computations over a unified execution engine while providing scalability and fault tolerance. One key concept of Ray is providing distributed components in its architecture, for instance, a distributed scheduler [39].

Recently, multiple distributed stream processing systems have emerged to process large inputs of data streams in a distributed manner. Spark is one of these systems that provides a unified computing engine for batch and streaming data as well as providing high-level APIs for machine learning, graph processing, stream processing, etc [50]. Spark's architecture consists of a central cluster manager and worker nodes that execute the functions, Spark also adopts Resilient Distributed Datasets (RDDs) as the programming abstraction which are partitioned across the cluster and provide fault tolerance to Spark in the form of “Lineage” [51].

Flink is another stream processing distributed system that will be discussed in detail later in this section. As well as Kafka which presents a distributed message broker that handles large amounts of data streams. These systems give good examples of different key concepts we look to have in our Piccolo system.

### 3.2.1 Flink

Apache Flink is a distributed processing engine for processing stateful computations over bounded and unbounded data streams [52]. It presents a unified architecture for stream and batch data processing [53]. It is considered a data analytics framework that provides data processing over a large scale of input streams while providing insights in real time.

The Flink software stack consists of four main layers: Deployment, Core, APIs and libraries. Flink can be deployed locally, in a cluster, or in the cloud. The core of Flink is a distributed data flow engine that executes a program created by different APIs as a DAG of stateful operators connected with data streams with support for batch processing and stream processing. For this, Flink offers DataSet and DataStream APIs respectively. Flink also provides libraries for machine learning, graph processing, and SQL-like operations [53].

**3.2.1.1 Architecture** As shown in Figure 5, Flink cluster includes three main processes: the Client, the Job Manager, and the Task Manager.

- **The Client**: Transforms the submitted code into dataflow graphs and submits it to the Job Manager.
- **The Job Manager**: It is the central orchestrator of the system. It schedules task execution, coordinates the checkpoints, and handles recovery upon failures. It consists of three different components: the Resource Manager, the Job Master, and the Dispatcher. There is one Resource Manager in a Flink cluster, it schedules the resources in the cluster and interconnects with external scheduling systems, i.e., resource management systems such as Kubernetes, Yarn, etc. Every running instance of Flink’s application is managed by one dedicated Job Master.
- **The Task Manager**: It is the process that performs the actual data processing. It is a JVM process that executes the tasks and exchanges data streams. It accepts tasks to process according to the number of its task slots [54].

**3.2.1.2 Scalability** In systems like Flink, scaling means adding more worker nodes to the system and thus increasing the parallelism of a submitted job. This scaling comes with some challenges

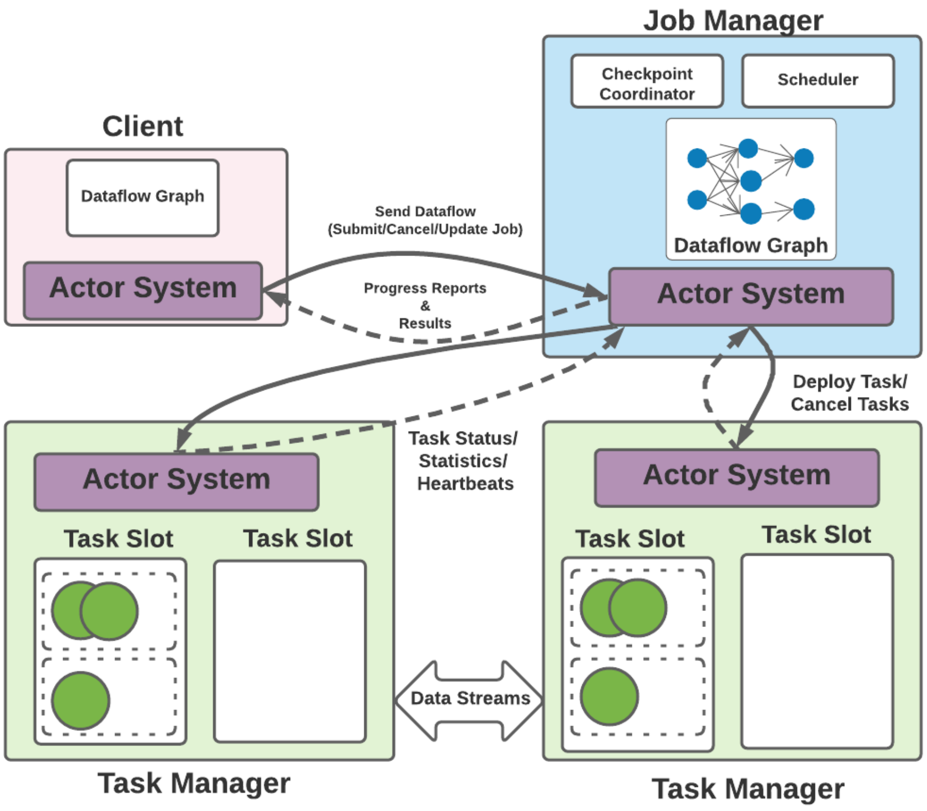


Figure 5: Flink Cluster

to be solved. One important aspect to think about when considering system scaling is the "state". Scaling the system to a new parallelism should handle the previously computed state and redistribute it in more or fewer partitions across different numbers of parallel operators. This must be done in a consistent and meaningful way.

Flink provides checkpoints and savepoints to share the state in the system. Other than achieving fault tolerance, this allows Flink to rescale the application by redistributing the state across parallel instances. After implementing checkpointing, Flink restores with the new parallelism.

To provide redistribution of the state, Flink modifies the checkpointing interface into ListCheckpointed. You can see the difference in Figure 6. Using this interface results in a list of state partitions and not a single state and this helps in redistributing it later. So using a list makes each item in it an independent, redistributable part of the state [55].

Currently, Flink has no support for autoscaling, i.e., it needs additional frameworks. To perform a scaling operation, the job must be stopped, checkpoints must be created, and then the system is rescaled.

**3.2.1.3 Fault Tolerance** Flink offers fault tolerance by recovering the state of the data stream processing application with exactly-once semantics. This state recovery is handled by a mechanism that periodically takes a snapshot of the state and stores it at the master node or HDFS. Thus, in case of

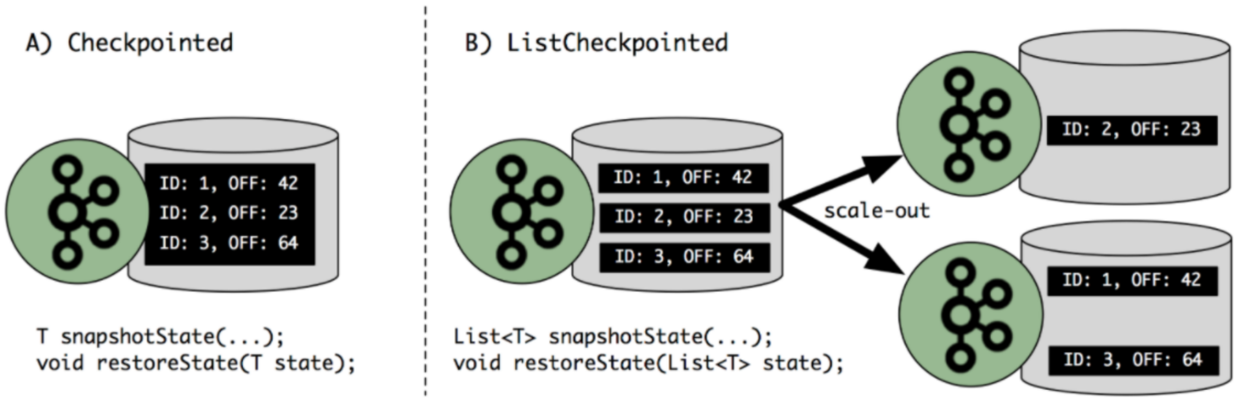


Figure 6: Flink ListCheckpoint [55]

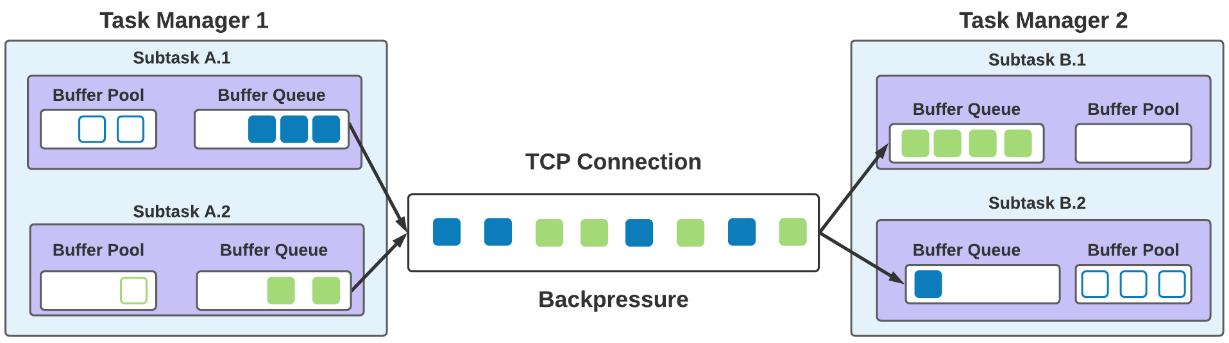


Figure 7: Flink BackPressure

failures, Flink stops the data processing and restarts from the latest checkpoint while the streams are reset to the point this snapshot was taken.[56]

**3.2.1.4 Network** Flink uses RPCs via Akka [57] for the coordination between the Task Managers and Job Manager. However, for the communication between Task Managers where the data streams flow occurs, Flink relies on lower-level API using Netty.

Tasks across different Task Managers communicate over TCP channels. And to reduce the resource usage, the network connections of subtasks belonging to the same task and scheduled on the same Task Manager toward one Task Manager are multiplexed over a single TCP channel. However, as shown in Figure 7, when a receiver subtasks's buffer is full and it cannot process any incoming records, it will stop consuming records from the multiplexed channel and thus results in backpressure for all other subtasks that are using this channel even if they have enough buffers to handle their incoming data.

Flink introduced a new mechanism to solve this issue. It implemented the credit-based flow control shown in Figure 8 where receivers will notify the senders about the available buffers they have as ”credits” and the senders will monitor the credits and forward buffers to the lower network stack in case of available credits.

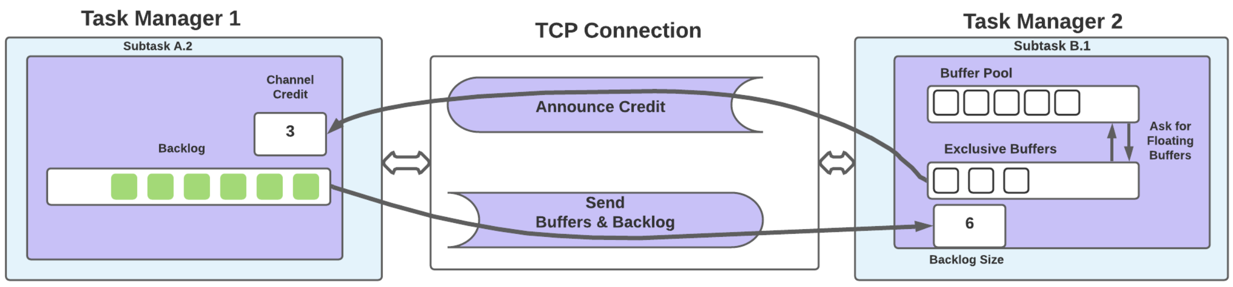


Figure 8: Flink CreditControl

Compared to the backpressure mechanism in earlier Flink versions, credits provide more control and the backpressure will be on one logical channel only and not over the multiplexed TCP channel. This will result in better resource utilization and better checkpoint alignments. [58] However, there are still some drawbacks to this mechanism as there are additional credit-announce messages. In addition to having some exclusive buffers not being used. And in case the credit announcement is slower than the data production, data will take more time to be sent.

Flink is one of the multiple distributed systems we are considering in this project. Being specialized in stream processing, Flink handles stateful and stateless functions over large amounts of data streams while providing fault tolerance. It serves as a good example in this scope. Looking at it in details, Flink has some interesting approaches especially in its fault tolerance mechanism, and how it handles the state during runtime, e.g., how it redistributes the state after the system scales or restarts. These are important concepts that are essential to every distributed system and they are directly related to our scope of work.

### 3.2.2 Kafka

Kafka is defined as a distributed messaging system [59]. It adopts the publish/subscribe messaging pattern. Kafka servers referred to as brokers, store the incoming messages pushed by the producers in the form of topics and partitions and consumers then subscribe to these topics and pull messages from the brokers [59]. Kafka serves the decoupling of producers and consumers of data in a system.

**3.2.2.1 Data Classification** Messages in Kafka are classified into topics, further, these topics are split into partitions. Partitions of a single topic can be hosted by multiple brokers allowing a topic to scale horizontally across the cluster and by using partitions and their replicas, Kafka provides redundancy and scalability.

As shown in Figure 9 there is the Leader replica and the Follower replica. The Leader replica is responsible for handling all produce and consume requests to a certain partition. Whereas the Follower replica just copies the data of the leader and keeps in sync with it. It is not responsible for answering any request sent to this partition. [60]

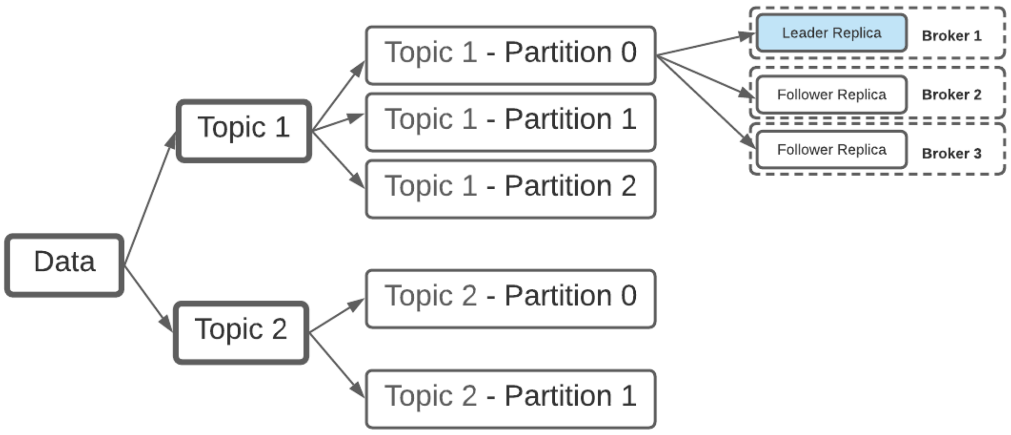


Figure 9: Data Classification in Kafka

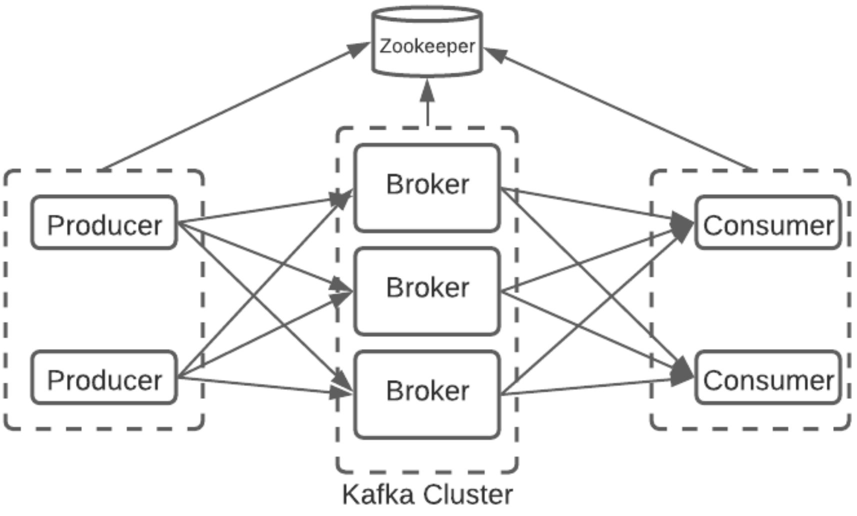


Figure 10: Kafka Architecture

**3.2.2.2 Architecture** Kafka architecture shown in Figure 10 includes Kafka servers referred to as Brokers and Kafka clients of two types: Producers and Consumers. Also, Kafka uses Zookeeper [61] to save metadata requested by different components of the cluster across runtime.

- **Producers**: the producer is the Kafka client that publishes data to topics, more specifically to their partitions. Producers also assign the records to a specific partition according to some defined keys or in a round-robin fashion to achieve load balancing.[62]

- **Consumers**: Kafka consumers subscribe to Kafka topics and pull the messages belonging to these topics. To be able to keep up with the high publish rate for producers, consumers usually work in groups. The consumer group which belongs to one application is defined by a set of consumers that work together to consume a topic by assigning each consumer of the group a part of the topic's partitions. So consumers in a group jointly consume the desired topic as shown in Figure 11. Kafka scales the data consumption by adding more consumers to a consumer group.

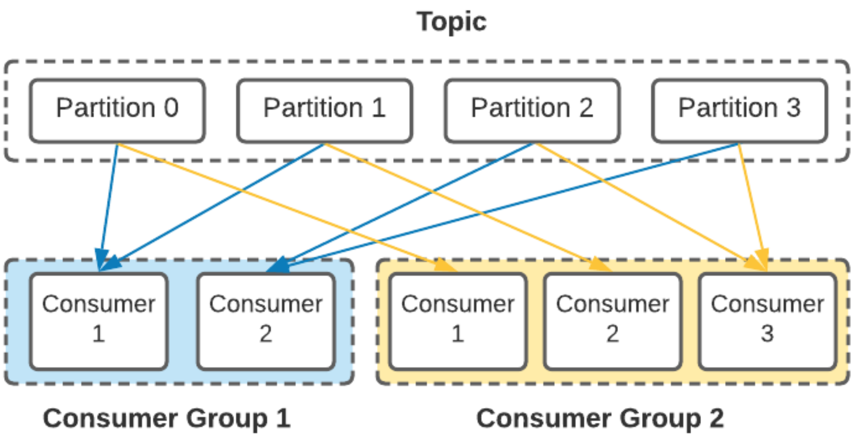


Figure 11: Kafka Consumer Group

This means that it is better to increase the partitions per topic.

These groups are managed by one of the Kafka brokers called "Controller". It monitors the consumers in the group and triggers a rebalance operation (for the set of partitions per consumer) in case of failures or the addition of consumers in the group. Consumers use the "offset" (another part of the metadata in Kafka) to save information about the messages they have consumed so that they can restart from the correct place. These offsets are unique per message in a partition and saved either in Kafka or in Zookeeper. [60]

- **Brokers**:Brokers are Kafka servers that handle Kafka client's request so it receives messages from the producers and handles them to consumers when receiving a partition fetch request. Brokers operate within clusters with one broker being the controller. This controller assigns topic partitions to certain brokers and monitors brokers' lifetime. Also, an important property in Kafka is data retention where the brokers keep topic data for a certain defined duration or until they reach a defined size. [60]

- **Zookeeper**: Kafka supports decentralized coordination of consumers without the need for a central master node. Instead, it uses a consensus service, Zookeeper. Mainly Zookeeper is used to detect the addition or removal of brokers and consumers, Then it triggers a rebalance process. It also keeps the consumption information and keeps track of the consumed offset per partition. So using Zookeeper, consumers will be notified of changes in broker and consumer groups. [59]

**3.2.2.3 Fault Tolerance** Kafka provides fault tolerance using the replication concept mentioned above. Replication gives Kafka the ability to guarantee availability and durability in case of node failure. Kafka replicates a single partition across multiple brokers to achieve redundancy. So in case a broker storing the leader replica of a partiton fails, another follower replica will then handle the requests toward this partiton. So these replicas present a backup in case of leader failure.[59]

Kafka handles the coordination of a large number of messages between producers and consumers. It implements this in a scalable and fault-tolerant approach as described above. It introduces another model of how different distributed systems spanning various use cases provide these properties.

## 3.3 Distributed Programming Models

Piccolo has a strong focus on a fundamental architecture for a distributed in-network computing system and strives to support existing programming models. This section describes two generic models for distributed programming. IEC 61499 [63] is an international standard for distributed Programmable Logic Controller (PLC) programming and Lasp is a distributed programming model based on Conflict-free replicated data types (CRDTs).

### 3.3.1 IEC 61499

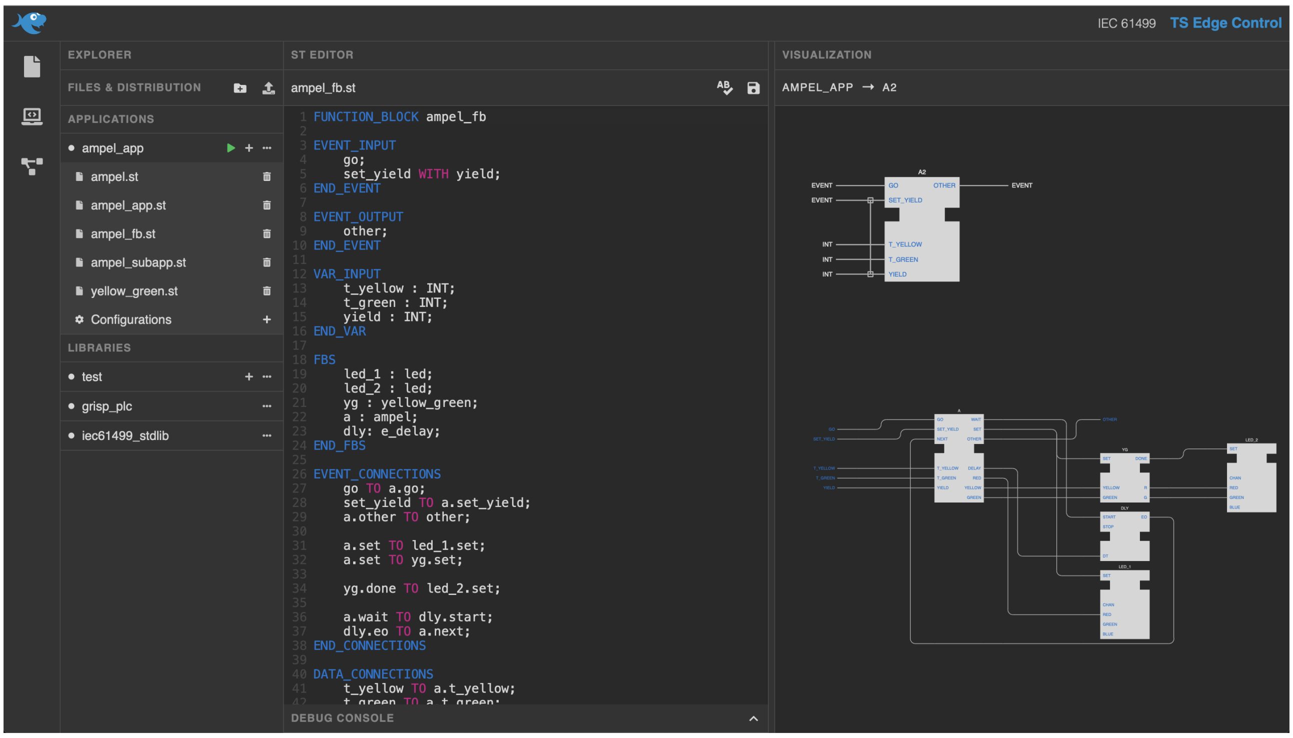
IEC 61499 [63] is an international standard for industrial process measurement and control systems based on IEC 61131 [64]. It defines a generic model and a system level design language for distributed systems aiming at portability, reusability, interoperability and reconfiguration of distributed applications. It targets in particular an environment for embedded devices, resources and applications and has an asynchronous event driven execution model.

In the larger picture, IEC 61499 is supposed to be the key technology for the transition to 'Industry 4.0' which includes large-scale machine-to-machine communication and the Internet of Things being integrated for the purpose of increased automation.

The basic function block is the fundamental component of an IEC 61499 application. Applications have a layered design as a network of subapplications, composite function blocks and basic function blocks with event and data connections. Subapplications again are a network of function blocks. Composite function blocks can contain other composite function blocks as well as basic function blocks. Each function block provides an interface of event I/O's and data I/O's. The data flow in an application is attached to the event flow, by assigning events with data inputs and outputs using additional constraints.

Basic function blocks execute an event driven execution control chart (ECC), which is a state machine. Incoming events are triggering (possibly multiple) transitions within the state machine. Every state can have several execution control (EC) actions which are executed after a transition. An EC action identifies at most one algorithm and at most one outgoing event. An algorithm can use the input data attached to the event that triggered the transition and set the output data that will be sent with the outgoing event of the action. Transitions can be conditional with reference to the input data.

The IEC 61499 standard can be implemented in structured text (ST) defined in IEC 61131-3 [65] and be visualized using data flow graphs consisting of function blocks as nodes and event or data



```
FUNCTION_BLOCK ampel_fb

EVENT_INPUT
    go;
    set_yield WITH yield;
END_EVENT

EVENT_OUTPUT
    other;
END_EVENT

VAR_INPUT
    t_yellow : INT;
    t_green : INT;
    yield : INT;
END_VAR

FBS
    led_1 : led;
    led_2 : led;
    yg : yellow_green;
    a : ampel;
    dly: e_delay;
END_FBS

EVENT_CONNECTIONS
    go TO a.go;
    set_yield TO a.set_yield;
    a.other TO other;

    a.set TO led_1.set;
    a.set TO yg.set;

    yg.done TO led_2.set;

    a.wait TO dly.start;
    dly.eo TO a.next;
END_CONNECTIONS

DATA_CONNECTIONS
    t_yellow TO a.t_yellow;
    t_green TO a.t_green;
```

Figure 12: Screenshot of Stritzingers IEC 61499 IDE with a composite function block

connections between two function blocks represented as edges. Figure 12 shows a screenshot of a composite function block that is part of an example IEC 61499 application implemented in ST in the editor with a visualization on the right side.

To summarize, IEC 61499 provides:

- a distributed programming language,
- a generic modeling approach for distributed control applications,
- a modular design based on function blocks, and
- the management of data and event flows.

### 3.3.2 Lasp

Lasp is a distributed programming model based on Conflict-free replicated data types (CRDTs) introduced in [66]. The concept of CRDTs was formally defined in [67]. Each node has a replication of the data and operates on it. This allows concurrent distributed programming independent of persistent network connections. Lasp defines functional programming operations on CRDTs, which provide the concepts to resolve inconsistencies in the data. This is done by merging operations that assure that all replicas converge to the same.

The name Lasp refers also to the implementation of this programming model in Erlang [68]. Erlang has built-in functionality for building distributed systems, generally referred to as Erlang distribution (See Section 3.5.1). Lasp is a suite of libraries that provide a different API and an alternative method of creating distributed systems. It implements CRDTs and provides functions to operate on them. This way the programmer does not need to worry about data replication or messaging.

## 3.4 Compute-First Networking

Domain-specific distributed computing languages like LASP have gained popularity for their ability to simply express complex distributed applications like replicated key-value stores and consensus algorithms. Associated with these languages are execution frameworks like Sapphire and Ray that deal with implementation and deployment issues such as execution scheduling, layering on the network protocol stack, and auto-scaling to match changing workloads. These systems, while elegant and generally exhibiting high performance, are hampered by the daunting complexity hidden in the underlay of services that allow them to run effectively on existing Internet protocols. These services include centralized schedulers, DNS-based name translation, stateful load balancers, and heavy-weight transport protocols.

Compute-First Networking (CFN) [69] is based on the assumption that, especially for compute functions in the network, it is beneficial to design distributed computing systems in a way that allows for a joint optimization of computing and networking resources by aiming for a tighter integration of computing and networking. For example, leveraging knowledge about data location, available network paths and dynamic network performance can improve system performance and resilience significantly, especially in the presence of dynamic, unpredictable workload changes.

The above goals can be met through an alternative approach to network and transport protocols: adopting Information-Centric Networking as the paradigm. ICN, conceived as a networking architecture based on the principle of accessing named data, and specific systems such as NDN and CCNx have accommodated distributed computation through the addition of support for remote function invocation, for example, in Named Function Networking, NFN [70] and RICE, Remote Method Invocation in ICN [71] and distributed data set synchronization schemes such as PSync [72].

CFN 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. Its main features are:

1. CFN marries a state-of-the art distributed computing framework to an ICN underlay through RICE, Remote Method Invocation in ICN. This allows the framework to exploit important properties of ICN such as name-based routing and immutable objects with strong security properties.
2. CFN adopts the rigorous computation graph approach to representing distributed computations,

which allows all inputs, state, and outputs (including intermediate results) to be directly visible as named objects. This enables flexible and fine-grained scheduling of computations, caching of results, and tracking state evolution of the computation for logging and debugging.

3. CFN maintains the computation graph using Conflict-free Replicated Data Types (CRDTs) and realizes them as named ICN objects. This enables implementation of an efficient and failure-resilient fully-distributed scheduler.

4. Through evaluations using ndnSIM simulations, it has been demonstrated in [69] that CFN is applicable to range of different distributed computing scenarios and network topologies.

For Piccolo, CFN is relevant because:

- it is an example of an ICN-based distributed computing environment that leverages the typical ICN benefits (name-based access to authenticated data and computation results, in-network caching, no notion of location-based addressing);
- it provides dynamic general-purpose distributed computing without centralized resource management; and because
- it demonstrates the idea of joint optimization of networking and computing resources by supporting function instantiating close to input data and by providing a late-binding function invocation request forwarding scheme.

## 3.5 Orchestration & Management

This section describes some aspects of orchestration. Orchestration concerns that configuration and management of computing systems, and in the context of Piccolo includes that distribution and placement of functions across a distributed environment.

In both Cloud and NFV (Network Function Virtualisation), control of the allocation of resources as well as the in-life management of a service uses centralised management and orchestration (MANO) and virtual infrastructure management (VIM). Current open source projects developing MANO systems are OSM and ONAP, while OpenStack has become the dominant open source VIM, Kubernetes is set to become the dominant cloud native infrastructure manager. Being centralised gives these systems a full view of all they control which somewhat simplifies their development. However the centralisation also imposes a significant number of compromises and constraints: the centralised model does not readily scale; the MANO/VIM must be involved in all life-cycle, resource and in-life decisions; and a centralised MANO/VIM becomes a processing bottleneck which limits scaling.

MANO/VIM systems are currently capable of some level of platform awareness whereby the MANO/VIM know specific details of the hardware resources which go beyond normal abstractions, for example, understanding NUMA architectures and SR-IOV capabilities of servers. NFV has added enhanced platform awareness (EPA) to the standard cloud hardware abstractions to improve performance but this further limits scalability of MANO/VIM systems by adding to the complexity of centralised information. Cloudlightning developed the concept of a hierarchy of self-managing agents to manage resources in clouds but assumed a classical data centre architecture of an underlying VIM which does not scale for in-network compute. A further complication of using OpenStack or Kubernetes outside the datacentre as is required for NFV is that both OpenStack and Kubernetes are designed to run be in the same secured networking environment as the resources it manages (i.e., within the same datacentre) which means that separating OpenStack and Kubernetes from its resources is a potential security liability. The ready solution is to require a full instance of OpenStack even in the smallest of edge nodes, which makes a very inefficient use of the edge compute resources.

The scalability, security and efficiency limitations of VIMs like OpenStack are well known and some network operators are working to a 100 compute nodes per OpenStack controller limitation. With thousands of sites each with many compute nodes, the VIM overhead would be very large. Managing the lifecycle of the VIM itself, e.g., updating the VIM software, becomes a significant operational challenge. It is expected that Kubernetes is much more scalable than OpenStack but still not scalable enough for tier 1 network operators. Kubernetes depends on writing updates to a central database, which although distributed for resilience and read performance will still be a bottleneck in a large network, especially one supporting microservices. The current state-of-the-art concentrates on the orchestration and infrastructure management, leaving open the opportunity for new work to improve the translation of business strategy and objectives into orchestration policy.

SDN (Software Defined Networks) technology enables the network configuration to be under programmable software control and so to respond to changing requirements and environments much quicker. SDN on its own does not allow the network to be a general programmable computer nor does it optimize traffic control in itself. There are many forms of SDN technology (Openflow, Netconf/Yang, P4, SRv6pipes, SR6v6eBPF, etc.) that Piccolo can exploit to route function requests or remote invocations to the optimum in-network compute.

NFV (Network Function Virtualisation) can be considered an established technology. There is a set of standards from ETSI's NFV ISG that are being accepted by major manufacturers and open source initiatives. Moreover, 3GPP is also converging to the same concepts/standards. NFV has successfully replaced the manual installation and provisioning of network processing hardware with software, but provisioning is still done on fairly long timescales. Additionally, processing is now based on DPDK which means processor spinning even when there is no traffic to process—this is highly inefficient.

The rest of this section considers further a couple of aspects: allocation of resources through an auction mechanism, and runtime management state-of-the-art technologies.

### 3.5.1 Distributed Erlang

Erlang/OTP provides its own rather simplistic mechanisms for managing distributed applications, often referred to as distributed Erlang. A distributed Erlang system consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a node. Message passing between Erlang processes at different nodes is transparent due to an integrated mechanism for translating symbolic node names to machine addresses and a distributed process identification management. In a system with several nodes it is possible to orchestrate applications in a distributed manner. There are two primitive operations connected to this concept: failover and takeover. When a node goes down it is possible to migrate the application(s) from that failing node onto another running node. This is called failover. Further if a node is started then the application can be migrated to the new node. This is called a takeover. The migration is decided on priorities which are configurable on a per application basis. Note that it is the control of the applications that is distributed. [73]

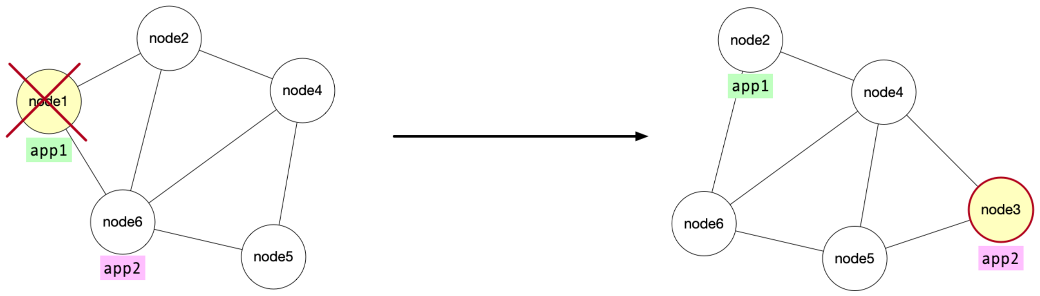


Figure 13: Failover of app1 and takeover of app2

The management mechanisms for distributed Erlang are pluggable in the sense that custom protocols and semantics can be implemented. This makes it a good foundation for concepts derived from Piccolo.

### 3.5.2 Resource Allocation via Auctions

There are multiple mechanisms to provision resources with different properties and based on various requirements and environments. Current research efforts leverage from the market domain and propose mechanisms that allocate resources via auctions at the edge and middle-tier locations of the network in order to meet demanding requirements (e.g., latency, bandwidth) of applications. An auction-based resource allocation and provisioning mechanism like Edge-MAP [74] perceives the resources of In-Network Computing Providers (INCPs) as a pool of interconnected virtualized hardware offered via independent marketplaces located at each cell of the network. In such a context, INCPs lease their resources to host bidding applications provided by Application Service Providers (AppSPs) and eventually get compensated by them.

For the AppSPs the incentive is the provision of higher QoS to their customers while for the INCPs the flexible management and expansion of their infrastructure. Computing resources are provided in the abstract form of virtual machines (static VMs, Containers, Unikernels), while a novel approach of the auctioning scheme is that the resources physically located but under-utilized at some region/cell can be advertised to neighbouring cell markets where demand exceeds supply.

Provisioning applications over a geo-distributed set of heterogeneous edge nodes differs from typical application provisioning in the sense that the allocation of resources needs to take into consideration the impact of network conditions and the load of each node. Edge-MAP adopts the notion of on-demand provisioning in which a VM is instantiated upon a client's request and for the duration of the client's engagement. The process of on-demand function provisioning consists of three phases, namely, resource discovery, resource allocation and resource configuration. In the competitive environment of INCPs the detection and allocation of resources (i.e., VMs) takes place in individual markets, located close to base stations or in the core network. Therefore, each INCP is associated with a market where its resources (in the form of VMs) are auctioned [75, 76].

The main features of Edge-MAP are:

- A polynomial time market mechanism for low-latency application provisioning over an edge-computing infrastructure taking into account end user mobility.
- On-demand provisioning of resources based on application requirements (especially effective in mobility scenarios) and usage of Vickrey-English-Dutch (VED) [77] auctions [3] which are ideal for repetitive allocation environments where they can leverage from the previously found equilibrium [4] for decreasing their execution time (resource allocation and discovery time overhead reduction).
- Provision of VMs to distant markets, which benefits INCPs in terms profit generation since they have the opportunity to diversify their prices.
- Scalable resource allocation mechanism as presented in [74] with special care in the challenging case of mobile users.

For Piccolo, Edge-MAP is relevant because:

- it provides a novel resource allocation mechanism tailored to the on-demand provisioning of applications with demanding requirements for end users.
- it provides resource allocation and discovery time overhead reduction since a customer request is accessing immediately a local market (e.g. on a cell-based approach or a LAN region), while simultaneously the number of bidders is considerably smaller which leads to lower execution time.
- it leaves open space for improvement in terms of security and privacy since the overall bidding procedure assumes truthful bidders. That is the bidding entity (AppSP) acts truthfully and bids for resources by just setting the bids equal to the actual gain of QoS of the application. Piccolo could further enhance it by providing security mechanisms to detect or block malicious bidders from injecting bid values that would break the market's equilibrium.

[3] This type of auctions facilitates the derivation of the unique minimum competitive equilibrium prices [78], commonly known as Vickrey-Clarke-Groves (VCG) prices; bidders cannot acquire their assigned resources for a lower price in any other competitive equilibrium.

[4] At this equilibrium price, the resources supplied is equal to the resources demanded

## 3.6 Secure Execution Environments

Besides the orchestration and management of resources, mechanisms to ensure security requirements of applications are crucial. This sub-section provides an overview of runtime isolation concepts of Linux containers, as well as secure container executions via SCONE.

#### Runtime Isolation of Linux Containers

Container-based virtualization enables the deployment and execution of distributed applications on cloud, edge/fog and Internet-of-Things platforms. Multi-tenant environments usually use Linux Containers [27] for performance isolation of applications and operations, Docker [28] for packaging the containers and Docker Swarm [79] or Kubernetes [80] for deployment and orchestration.

Under the hood, the container progression is based on two key pillars, linux namespaces and Linux Control groups (cgroup). The namespace produces a virtually isolated user space and provides to an application its dedicated system resources like process and user id, file system and network stack. Such an abstracted userspace allows each application to be executed independently without interfering with the rest of the applications on the same host. The cgroup administers hardware resources (CPU, memory, network and disk) limitation, prioritization and controlling of an application. Cgroups and namespaces are two fundamental building blocks that also provide the main differentiation between virtual machines and containers as the former are a hardware-level virtualization and the latter are an OS-level virtualization.

Even though containers outperform Virtual Machines on Hypervisors in terms of performance (faster startup times and increased I/O throughput), they do have a weak spot when it comes to security properties because the host OS kernel needs to protect a larger interface and in most cases only software isolation mechanisms are used [81]. In general, virtualized hardware isolation offers a much stronger security boundary than namespace isolation.

Existing container isolation mechanisms concentrate on protecting the environment from being accessed from untrusted containers. Tenants, however, require the protection of confidentiality and integrity of their application data from accesses by unauthorized entities that range from other containers to higher-privileged system software such as the OS kernel and the hypervisor. In general, malicious parties target vulnerabilities in existing virtualized system software or even compromise the administrators' system credentials.

#### Secure Containers (SCONE)

An approach to overcome such security issues is to combine hardware-level mechanisms and containers to protect user-level software even from privileged system software. Trusted Execution Environment (TEE) is a hardware-level mechanism that enables the execution of code and data inside tamper-resistant processing environments (commonly referred as "enclaves") and guarantees the authenticity of the executed code, the integrity of the runtime states (e.g., CPU registers, memory and sensitive I/O) and the confidentiality of its runtime code and data stored on a persistent memory. The most mature of such integration efforts is Secure Containers, commonly known as SCONE [82]. SCONE is a secure container framework that leverages from the SGX trusted execution support of Intel Skylake CPUs to protect container processes from external attacks. It supports Docker containers and Kubernetes as an orchestration service making it a suitable fit for microservice architectures. SCONE provides the abstraction of a secure container that makes it indistinguishable from a regular Docker container while it enriches a microservice with confidentiality and integrity. Each secure container executes a microservice instance inside Intel SGX enclaves. SCONE leverages from Kubernetes concepts and extends them using more fine-grained support in terms of security. Thus, SCONE forms Confidential Computing (CC) applications that contain confidential services (service's process running inside Intel SGX enclaves) that relate to Kubernetes pods[5]. All communication between confidential services is encrypted no matter in which pod each one runs. Another feature of SCONE is the configuration and attestation service (CAS) that enables not only seamless (without source code changes) attestation of services and access control to service's secrets but more importantly peer-to-peer attestation of services operated by mutually distrusting peers.

SCONE has the following properties that Piccolo can make use of:

**Small Trusted Computing Base (TCB):** SCONE exposes a C standard library interface to container processes using statically linking against a libc library (musl) within the enclave. To execute system calls the execution flow needs to be transferred outside the enclave (untrusted domain). To handle such transitions, SCONE encrypts/decrypts application data on a per-file-descriptor basis, which means that files outside of the enclave are encrypted and all network communication is shielded by TLS.

**Low Overhead:** Enclave transitions in general cost (e.g., I/O operations that need to exit the enclave) in terms of performance. To handle that, SCONE provides user-level threading that amplifies the time that threads spend inside the enclave.

**Transparent to Docker Containers:** Secure containers behave the same as regular containers in the view of the Docker Engine. SCONE requires only an SGX-capable Intel CPU and an SGX kernel driver.

**Seamless Attestation:** Ensures (without source code changes) that an an application indeed is executed inside of an enclave. Secrets (e.g., configuration files, keys, arguments, environment variables) are provided to the application after a successful attestation procedure with a remote trusted entity. The distribution of keys is integrated to SCONE and no application changes are required.

[5]Kubernetes Pod: the minimal deployable unit consisting of one or more containers executed on the same node

# 4 Initial Architecture

Historically, edge- and in-network computing has been perceived as plumbing pipes to servers or as intercepting traffic for applying some form of computation. Piccolo takes a more principled approach that is challenging these system designs.

Piccolo's approach to in-networking computing is leveraging fundamental distributed principles (such as the ones described in section 2) for enabling distributed application development and deployment across a wide range of platforms and network configurations.

We use the term *architecture invariants* to describe important architectural pillars (as opposed to a complete system architecture that would be implemented in a waterfall model) that we deem important for guiding further research and experiments.

This section describes Piccolo's initial architecture invariants and system properties that will be the basis of the evolving Piccolo system architecture. In the following sections, we discuss different relevant invariants/properties, using the decomposed distributed Vision Processing scenario as depicted in Figure 14 for illustration.

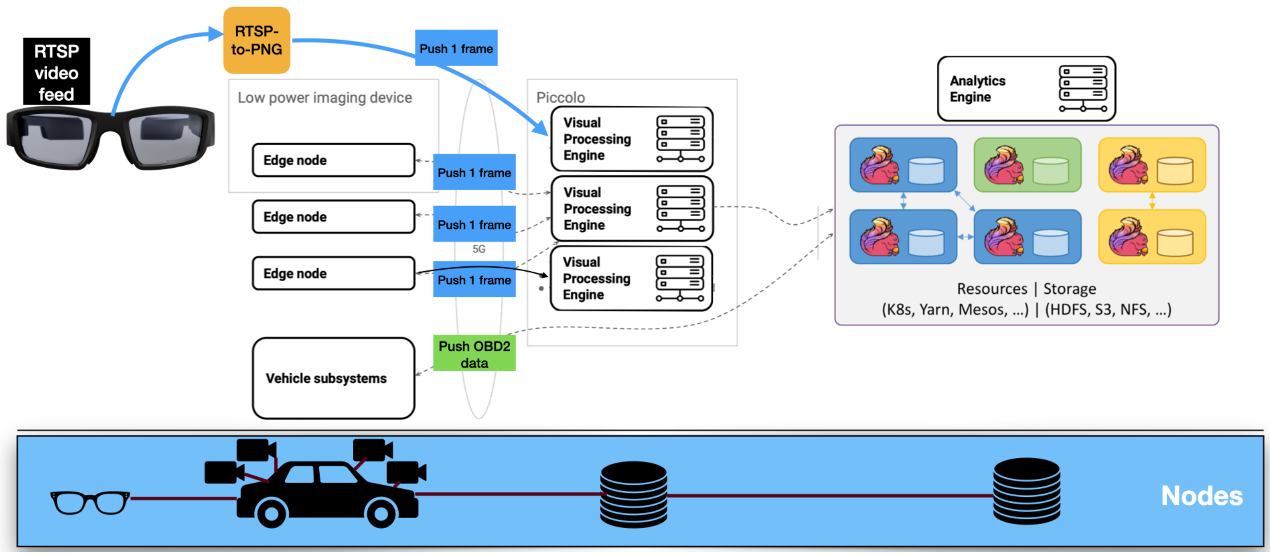


Figure 14: Vision Processing Decomposed

Figure 14 depicts an instantiation of a decomposed vision processing application with several video inputs (AR goggles, Edge nodes), three instances of a Video Processing engine (applying ML-based image processing), and a (further-decomposed) Analytics Engine.

The different instances are running on different platforms (Android-based AR goggles platform, car camera platforms, edge computing platforms and cloud platforms).

## 4.1 General-Purpose Distributed Computing

The vision-processing use-case example utilizes the stream processing paradigm (see Section 2.2.1) such that computational tasks can be organized in pipelines across a network. While the distributed computing capabilities of a Piccolo system will certainly cover such a use-case it should be explicitly noted that Piccolo is targeting a much broader class of distributed computational problems.

In Piccolo, the network is a computing platform with support for arbitrary distributed algorithms and applications. This means, in particular, that data does not need to be self-contained, control loops between nodes can be part of an algorithm and the representative dataflow network (see Section 2.2.1) abstractions are not necessarily free of cycles (which is usually not the case in stream processing). Piccolo nodes and functions are capable of holding state, and decision making for processing and sending data can depend on it. Message passing will be used as the fundamental model of communication since it allows to deal with distributed algorithmic complexity in an opaque way.

An example platform that approximately fulfills these requirements for a rather closed ecosystem is Erlang/OTP (see Section 3.5.1). The initial architecture described in this document should lead in the long run to an abstract API for distributed communication to enable general-purpose distributed computing in a Piccolo system.

## 4.2 Reliability and Consistency Architecture

A key feature of Piccolo is that it supports the execution of distributed functions. However, the execution of distributed functions has important differences to the execution of centralised functions which makes reliable and consistent execution more difficult to achieve.

A fundamental principle of Piccolo is that reliability and consistency should be established and assessed at the scope of the end user. Further, the Piccolo architecture and protocols should make use of the redundancy inherent in the distribution network as a key part of establishing reliable execution. Importantly, this principle extends beyond the scope of any one functional component (e.g., as achieved by Flink for reliable stream processing) and any one specific execution environment where reliability may be achieved through the use of specific features of the execution environment and infrastructure which are not available in the full scope of Piccolo infrastructure.

It has been well established over many years that some important assumptions that are made in the architecture and design of centralised systems do not hold for distributed systems. Famously, Abadi stated the “PACELC" principle that “Under Partition, either Availability or Consistency; Else under normal, either Latency or Consistency" [83].

This specifically identifies that maintaining consistency of state within a distributed system cannot be achieved without sacrificing other parameters. In a fully working system, there is always a trade-off between the consistency of the state across the system and the latency with which the consistent state can be accessed. If the distributed system becomes partitioned such that communication to some parts of the system becomes broken, consistent state is not available to all parts of the system.

Piccolo embraces rather than avoids these fundamental trade-offs. At this stage in the project, we set out a number of definitions and state three architectural principles which will be developed of the course of the project.

### 4.2.1 Piccolo Definitions of Function, Object, Stateless, Session, and System

These terms are widely used however, their exact definition can sometimes be ambiguous and depend on the context in which the terms are used.

For example, we set out describing Piccolo as a concept which is "stateless" and "serverless". This statement captures important aspects of the Piccolo architecture; however, it could also be misunderstood by some unfamiliar with the informal use of the terms in cloud computing. Indeed, according to definitions widely used in other contexts, Piccolo functions do hold state, Piccolo functions are themselves state (see below), and Piccolo nodes can be servers.

As is explained in more detail below:

- "Stateless" means that all state is held in the context of a session and the state terminates when the session terminates;
- "Serverless" means that functions are referenced by their function name and not by the network location of the compute platform hosting the function.

More specifically, function is a term that is used widely but whose definition varies considerably from a broad and loose definition to a very tight and precise mathematical definition. For the purposes of Piccolo, we will develop a precise set of definitions to cover the scope necessary for Piccolo and making distinctions where is this important. The following is a start point for this set of definitions.

- **Pure function:** This is a mathematical construction which maps a selection from a domain set to a selection of a range set where the domain set and range set can be tuple types. The function is called with the domain selection and returns the range selection and no state is held between function calls. No function call has any "side-effects", that is, a function call does not change any other state in the wider system. The return of the selection from the range set is the only consequence of a function call. The pure function is the basis of the functional programming paradigm and greatly improves testability as the behaviour of a pure function is so heavily constrained.
- **Stateless function:** A stateless function is like a pure function in that no state is held between

function calls, however, the stateless function will have "side-effects". The standard example is a function 'print'. The calling parameter is normally a file to be printed and the return to the call is normally 'completed' or 'failed'. However, the call will change the state of the printer elsewhere in the system; this is a side effect of the function call which is not part of the function return.

- **Process function:** While a pure function is defined by a mapping, a process function is described by a state machine. It is normally quite unlike a pure function and is normally oriented to operations which are more of a continuous flow rather than a single, discrete call event. A process function may receive inputs from several sources during the process operation. The process function holds the state it currently is in as part of its operation and may well hold other information state as well.

- **Session:** A session is a set of coordinated state held across several process functions. The state is held only for the duration of the session such that the duration of the state and the duration of the session are synonymous. A key Piccolo proposition is that all state is always part of some session (see below).

- **Session function:** This is a process function with a well defined start point and stop point where the process state machine starts and where it terminates. It is called a session function because in the context of Piccolo, any one process function will operate in the context of other process functions where the starting points and stopping points are coordinated and therefore constitute a session.

- **Actor:** An actor can hold state and will normally interact with asynchronous calls to methods. While an actor model may may be implemented in several ways (eg coroutines), the normal model of execution for actors in a actor model is that actors execute on independent threads of control and any one actor could have more than one internal execution thread. While actors and sessions have different origins, when an actor is considered in the context of actor lifecycle management, a concept of an actor and a session function converge.

- **Cyclical process function:** This is a process function where the process is repetitive, and the state of the process returns regularly to the same state in the state machine in the course of continuous operation, for example for stream processing. However, any cyclical process function must still have a start point and stop point will always be in the context of some wider session, so while the emphasis is on the cyclical nature of the process, this is still a session function.

- **Intermediate function:** An intermediate function is a form of stateless function which is important to Piccolo. An intermediate function will accept input calls and in response make its own calls to other functions. The intermediate function may hold state pending a response from these consequential calls and form a return to the original call dependent on the responses to the consequential calls. Alternatively, the intermediate function could 'drop out' from the wider session if the function does not hold any state associated with the session.

- **Object:** The object-oriented programming paradigm is distinguished from the functional programming paradigm by the presence of state. An object can hold state between any calls to methods belonging to the object. Functional programming seeks to avoid state completely, object-oriented programming seeks to secure state behind a set of well-defined methods. Some more modern languages bring the two together, for example Scala's support for stream processing brings together the concept of a session function with object-orientation.

This set of working definitions has not addressed the issue of inheritance and other ways of reusing code (rather than the real time reuse of live functions/objects that the code creates) to enable a 'write once' policy which underpins refactoring in software engineering. These definitions also do not consider polymorphism and abstraction of functions and/or objects. Both are important considerations and can create conflicting requirements. For example, the relatively recent language Julia explicitly rejects object-orientation and inheritance in order to achieve more efficient functional polymorphism.

At this stage, the primary focus of Piccolo is on the converging concepts of the actor and the session function. A session function is truly stateless outside of the context of any one session and therefore conforms to the stateless objective of Piccolo. However, it does also allow for state, but state must always be in the context of a session which will consciously and explicitly define the lifecycle of the state.

Extending on the session function definition, we set out three key architectural principles for Piccolo relating to state and consistency of state.

### 4.2.2 Generalised End-to-End Principle

In Piccolo, the reliability architecture is directly linked to the management of state in a way that is analogous to the end-to-end principle underpinning the Internet. From the point of view of the end points of an interaction, all intermediate execution between the end points is regarded as being unreliable and it is assumed that any state held by such intermediate execution may be errored or inconsistent. This is analogous to the principle of "soft state" held by Internet protocols such as TCP, PPP, and many routing protocols.

In terms of the Abadi principle, in Piccolo, intermediate execution will favour availability and latency over consistency. Consistency is only assured by self-checking on an end-to-end session basis. This has two important consequences:

- the end points of the distributed function must implement reliability and consistency measures and cannot regard reliability and consistency as something which can be delegated to capabilities within the infrastructure;

- Intermediate components do not need to be designed to guarantee reliability or consistency and can be optimised for latency and availability.

This contrasts with many current component systems where consistency is often a prime design feature which is achieved even as the component system scales to massive proportions (again Flink is a good example). This does not mean that such component systems cannot be used within Piccolo, it simply means that the Piccolo is not relying on the component reliability and consistency to achieve end-to-end reliability and consistency.

### 4.2.3 State is Always Owned by a Session

This principle can be illustrated by garbage collection which is also a special case of the principle. Execution environments with garbage collection keep track of all other objects which have knowledge of each object created within the execution environment. If an object is not known by any other object, it is considered garbage and deleted.

With garbage collection, the execution environment keeps track of the context of each object state. In Piccolo, all state is part of a session context which defines the lifecycle of the state. When a session terminates then so does all state associated with the session.

However, this is itself a good example of why distributed systems are different to centralised systems. With the garbage collector, the state of each object with respect of other objects can be treated as accurate and immediately consistent and this is fundamental to the reliability of garbage collection. With a distributed system, the session state as known by each distributed component may be inconsistent, for example, an intermediate component may be unaware that a session has been terminated by the end component.

There are several ways the unreliability can be accommodated, two important ones being:

- no intermediate state—the intermediate functions do not hold any session state and only respond to each call and then drop out the session;
- soft state—the intermediate systems set a life timer on the state which must be refreshed by the wider session or the state is deleted.

### 4.2.4 One Session's Static Function is Another Session's Dynamic State

In the classical definition of a system, the transfer function is static/unchanging while the state is dynamic/changing. This is also true in object-orientation. The runtime system will store the compiled methods as part of the programme code while the object state is stored in dynamically managed memory.

While many environments treat these as two distinct worlds, in Piccolo we treat these as being two different session contexts, as at an abstract and more fundamental level, they are the same.

For an intermediate function to be available to be part of a session, the code that executes the function needed to the loaded into the execution environment. In Piccolo we define this process of loading the code as being part of some management session. The code is then part of the dynamic session state of this management session which loaded function code and manages the lifecycle of the function code.

However, when a session at a higher level wishes to use this function as part of its session, the function code is assumed to be static and is not part of this session's state. The code defines the function's logic, that is the transfer function in system terminology, and is persistent between sessions at this level.

An example of this is in the Akka actor model where actors can participate in process sessions and hold state for process sessions. However, Akka actors also have their own lifecycle which is carefully managed by the Akka runtime environment. The actors are part of the dynamic state of the Akka runtime.

## 4.3 Actor-Model as a Fundamental Abstraction

As described in Section 2.2.2, when processing a received message, actors may modify their local state and therefore influence their future behavior [8]. Hence, actors are stateful and possess the expressive power of the process functions defined in Section 4.2.1. As one can assume their lifecycle is controlled by and bound to a parent—the node or execution environment executing them, one can, according to the definition presented above, classify actors as session functions. A pure function can be implemented by an actor never modifying the state, while a stateless function can be implemented by an actor causing a side-effect outside of the actor abstraction.

As all state is local to a single actor and therefore bound to the actor's (session function's) lifetime, a system adhering to the actor model complies with definitions of session and state presented in Section 4.2.1. The sole ownership of state by a single actor allows for precise reasoning about the lifecycle of all state held in the system.

As suggested in Section 2.2.2, the actor model provides a natural way to reason about the validity of state in scenarios such as a network partition or horizontal scaling of an application component. In the case of a network partition, the state held by an actor can only be interacted with (using messaging) if that actor is in the same partition. When there are multiple actors, either due to the network partition or the attempt to scale a computation horizontally, these actors are unable to share state and each instance needs to be addressed explicitly. Hence, the actor model provides a natural abstraction around the trade-offs between consistency and availability discussed in Section 4.2.

As the interaction between actors is only possible using messaging, for which the actor model introduces the requirement of knowing another actor's address [8], the actor model furthermore provides a natural abstraction for location-agnostic interaction and isolation boundaries.

Actors may be used to implement different high-level applications and interaction patterns. As an example, the request-response pattern may be implemented by sending a request message containing the requester's address and subsequently receiving a response message [9]. Stream processing pipelines as the ones supported by Flink may be implemented by instantiating a set of actors that together implement the dataflow graph.

In summary, actors may serve as the building block for distributed applications running on Piccolo and provide a natural abstraction for state and lifecycle rules, network transparency, and isolation.

## 4.4 Names, Namespaces, and Nesting

Piccolo enables nested construction of distributed computing systems by employing separate namespaces for (infrastructure) Nodes and Actors (functions). The Actors in one distributed computing context could represent (infrastructure) Nodes in a upper-level namespace.

For example, a network of VMs could run a particular Piccolo application, with the VMs representing Piccolo Actors and the hosts representing Piccolo Nodes (with the corresponding Agent functionality). These VMs could themselves be Piccolo Nodes in another namespace, e.g., for hosting container Actors (which again could be Nodes in yet another namespace, hosting JVM Actors etc.).

From a namespace perspective, these different different namespace are totally independent. Changes in the underlying infrastructure would not require name changes in the client network. Similarly, named Actors can be moved from one Node of an underlying infrastructure because there is no tight coupling between Actor and Node from an identity and namespace perspective.

In some network architectures, for example ICN networks, it would still be possible to leverage a single network infrastructure for the communication for all namespaces. In other network architectures, for example IP networks, it may be required to employ nested virtual networks.

Some systems are using internal mappings from (for example) an application-relevant namespace to lower-layer namespaces or address spaces. For example, mapping Actor names to TCP/IP endpoints and resolving Actor names by an orchestrator. Piccolo is going to explore different approaches in that context, including fundamentally name-based operation, where the Actor names are visible to the network and do not need to be resolved (or would be resolved in a late-binding fashion), which can enable joint optimisation (Section 4.7) and help with decentralising the system.

## 4.5 Piccolo Nodes and Actor Implementations

In Section 4.3, we proposed that actors could act as a basic building block for distributed applications running on Piccolo. These actors need to be executed on the Piccolo nodes. As suggested in deliverable 2.1 [2], a node might provide one or more execution environments that run a certain class of actors and use a subset of the resources provided by a node. These execution environments could be highly heterogeneous: Some of them might execute serverless actors with managed lifetimes and state, which are only scheduled to process incoming messages. Others might execute long-running containers housing multi-threaded applications, whose interactions are only partially managed by Piccolo and which might cause side effects unknown to Piccolo.

The execution environments and actors might have different resource requirements such as the required processing, memory, and storage capacity, virtualization technology, and access to specialized hardware. Furthermore, they require the actors to be specified in different formats, e.g., a container or WebAssembly binary. While each actor is executed by a single execution environment, one distributed application might be composed of a set of actors requiring different execution environments and node properties.

Applications might include multiple representations of the same actors for multiple types of execution environments, which potentially have different performance characteristics, resource requirements, or might only be present on a subset of the nodes, allowing the system to select the best execution environment based on the current situation. In a distributed and potentially mobile environment, this extends to a complex multidimensional decision about service placement, which Piccolo intends to solve.

In summary, a core challenge of Piccolo as a distributed system is the efficient use of compute and network resources across a heterogeneous set of nodes to run a set of heterogeneous applications comprised of actors. Due to this high degree of heterogeneity, Piccolo can be seen as an attempt to integrate the classical, application-oriented approach to cloud computing and the state-of-the-art serverless approach into a single platform that spans the entirety of the network.

## 4.6 Pluggable Protocols

Piccolo aims to provide a distributed execution system for a range of different applications and use cases, which use different interaction patterns. Examples for these patterns include the basic request/response semantics often employed by Microservices, Remote Method Invocation (RMI), which is by used by SOAs and systems such as Ray and CFN, stateful stream processing, which is used by systems such as Flink, and publish/subscribe. It is also worth mentioning that a single application might implement different types of interactions, for instance, one for control functions and the other for data exchange. Different applications and interaction patterns might require different delivery guarantees. These abstract patterns can be implemented using multiple protocols. Several applications with request/response interaction patterns and requiring ordered and correct delivery can be implemented using HTTP on top of TCP in traditional connection-oriented architecture. On the other side, Piccolo would implement extensive data exchange applications like stream processing with data location having less importance than the data itself in an ICN-based approach. Being able to execute on top of traditional connection-oriented or ICN-based architecture gives Piccolo the capability to optimize applications' implementation and achieve a more novel execution approach when possible.

Piccolo should support the applications in implementing these various interaction patterns by mapping them to protocols based on the requirements of the applications, nodes, and execution environments. However, while an automatic and flexible protocol binding might be possible and desirable for higher-level actors, existing or specialized application components might be required to communication with other, potentially static application components using a fixed interaction pattern and protocol binding. Hence, Piccolo should only offer its assistance in achieving communication across application components and not enforce the use of any abstraction or protocol.

To support this, applications should be able to specify the required interactions in an abstract way, and Piccolo should then bind these interactions to one or more communication protocols based on the current situation and constraints. As an example, RMI could be specified using an abstract definition similar to SOAP[84], which itself could be generated by a higher-layer abstraction such as Ray and CFN, which Piccolo then maps to either a connection-oriented protocol or an ICN-based approach such as RICE. Similarly, dataflow graphs could be mapped to various communication protocols supported by the environment. Finally, as suggested in 4.3, the different interaction patterns could be implemented using actors and actor messaging, reducing the problem to mapping messaging between actors to one or more communication protocol supported by the involved actors, nodes, and execution environments.

## 4.7 Joint Optimisation

Traditional in-network compute systems treat computing and networking as separate concerns. For example, in overlay-based distributed computing system such as Flink, a job manager makes decisions for task placement (possibly employing an orchestrator such as Kubernetes) and then establishes TCP connections between tasks (or task managers). In Mobile Edge Computing Systems, a traffic management system is responsible for mapping certain user requests (via traffic steering) to an edge proxy (instead of an origin server). In both approaches, connectivity is taken for granted, and network performance is, at best, only considered statically (for example when selecting an edge computing proxy).

**Joint Optimisation** refers to the concept of enabling a tighter integration of computing and networking (and storage), so that the distributed system can make better informed decisions (at run-time) with respect to the allocation of compute and storage resources, request forwarding and overall compute graph construction. The general concept has been illustrated by approaches such as *Jointly Optimal Routing and Caching for Arbitrary Network Topologies* [21] and systems such as DECO (Data-cEntric COmputation) [85].

Technically, a Piccolo system could leverage multiple information sources with different levels of accuracy, timeliness, dynamicity, and convergence properties as well as more static configuration such as developer, user, and operator requirements. For example, static information on general compute resource availability could be enhanced by dynamic run-time probing (or implicit measurements).

This general architectural approach also enables rethinking the separation of functionalities in networks and distributed system infrastructure. For example, network routing systems could be enhanced to provide some (potentially aggregated) overview of available compute resources and available Actor instantiations (including a cost abstraction). On a Node, this information could be transformed into static information on the reachability of other Nodes or Actor instantiations. On the data forwarding layer, forwarding functions could be enabled to take real-time observation on reachability, latency, and overall performance into account to select one Node or one Actor instantiation out of a set of candidates.

In the decomposed Vision Processing example as depicted in figure 14, the vision processing on the Vision Processing Engine (VPE) could be a stateless function that works on individual pictures. In such a setting, there may be knowledge in the system about the general existence of corresponding Actor instances, but the system could benefit from allowing for dynamic forwarding and instance selection based on current performance/load, reachability, cost etc.

In summary, Piccolo's concept of rethinking the integration of computing, networking, and storage is expected to enable new opportunities for building more flexible and better performing distributed systems, and joint resource optimisation is a promising key technology to achieve that.

## 4.8 Summary

The Piccolo architectural invariants are key differentiators to other systems – they can be summarised as follows:

**General-Purpose Distributed Computing**
Piccolo is targeting a broad class of distributed computational problems. The network is a computing platform with support for arbitrary distributed algorithms and applications. Message passing is used as the fundamental model of communication since it allows to deal with distributed algorithmic complexity in an opaque way.

**Reliability and Consistency Architecture**
Reliability and consistency should be established and assessed at the scope of the end user. The implication is that the end points of a distributed function must implement reliability and consistency measures (and not rely on the use of specific features of a particular functional component, execution environment or infrastructure, because they are not available in general). Further, the Piccolo architecture and protocols should make use of the redundancy inherent in the distribution network as a key part of establishing reliable execution.

**Actor-Model as a Fundamental Abstraction**
Actors may serve as the building block for distributed applications running on Piccolo and provide a natural abstraction for state and lifecycle rules, network transparency, and isolation.

**Names, Namespaces, and Nesting**
Piccolo distributed application programs are generally address-agnostic and work with names in an application namespace. Piccolo enables nested construction of distributed computing

systems by employing separate namespaces for (infrastructure) Nodes and Actors (functions). The Actors in one distributed computing context could represent (infrastructure) Nodes in a upper-level namespace.

**Piccolo Nodes and Actor Implementations**

Piccolo aims to efficiently use the available compute and network resources across a heterogeneous set of nodes to run a set of heterogeneous applications comprised of actors. Due to this high degree of heterogeneity of actors and execution environments, Piccolo can be seen as an attempt to integrate the classical, application-oriented approach to cloud computing and the state-of-the-art serverless approach into a single platform that spans the entirety of the network.

**Pluggable Protocols**

Piccolo aims to support a wide range of applications, thus supporting these applications' interaction types. According to the application's requirements, dataflow, request/response, RMI, and other interaction types can be implemented using different communication protocols. For instance, mapping dataflow to an ICN-based approach. And as Piccolo will be using actors and actor messaging for implementing these different interaction types, it will then map the messaging between actors to different communication protocols.

**Joint Optimisation**

Piccolo will enable a tighter integration of computing and network (and storage), so that the distributed system can make better informed decision (at run-time) with respect to allocation of compute and storage resources, request forwarding and overall compute graph construction. A Piccolo system could leverage multiple information sources with different levels of accuracy, timeliness, dynamicity, and convergence properties as well as more static configuration such as developer, user, and operator requirements.

# 5 Conclusion

Based on our analysis of current distributed computing trends (Section 2 and Section 3) and the use cases investigated by Piccolo partners (Piccolo D1.1), we have developed an initial Piccolo Node design (in Piccolo D2.1) and a set of architectural invariants, describing key differentiators and important principles for the overall system design. In the following, we summarise key properties, identified research challenges, and next steps for Piccolo.

## 5.1 Piccolo Properties

While the research and development of the platform is still ongoing, we have identified quite a few interesting opportunities. The key differentiating properties of the Piccolo platform as described in Section 4 are:

**Distributed Computing for Cloud and Edge**
Most distributed computing frameworks are designed for well-controlled, stable environments such as data-center networks, compute clusters etc. Piccolo aims at *enabling successful distributed computing patterns at the edge*, considering heterogeneous network characteristics, dynamic changes and different stake holders.

**General Purpose Distributed Computing**
Most distributed computing environments are targeting specific interaction paradigms such as stream processing (Flink) or message-based coordination (Kafka). Piccolo aims at supporting all relevant interaction types and at providing a *general-purpose distributed computing platform, fundamentally based on an Actor model as a basis for supporting different specific interaction types*. This include different levels granularity and different lifetimes for Actors, for examples from microservice components to stateless functions.

**Composability**
Piccolo Actors are black boxes for entities (e.g., other Actors) that instantiate them and interact with them. For example, an Actor could represent a certain Function, and calling that Function can result in a chain of additional Function invocations, i.e., the *interactions can be nested*, and Actors/Function can represent larger as well as very fine-granular units of computation.

**Multi-Platform**
Unlike most systems such as existing micro-service systems (based on Docker, Kubernetes), or application layer frameworks (Apache Flink, Spark, Kafka, based on Java), *Piccolo is fundamentally agnostic to hosting platforms*, and can support running Actors in bare-metal environments, virtual machines, containers, as processes in an operating system, or as threads in a JVM. This will also enable heterogenous deployments so that the most adequate platform can be chose for specific functions in a distributed system.

**Recursive Multi-layer Nesting**

The Multi-Platform approach and Piccolo's namespace concept (naming Actors independently from infrastructure) fundamentally enables nesting Piccolo systems so that one set of distributed Actors can form a Node infrastructure layer for an upper level of computing.

## 5.2 Piccolo Research Challenges

We envision a set of interesting research and development challenging for achieving these properties, including:

**Scalability and Failure Resilience**

Scalability and Failure Resilience (e.g., automatic fail-over) are typically relevant and important features in a distributed computing environment. We envision specific challenges due to the more flexible system architecture and potential heterogeneity in Piccolo system instantiations.

**Joint Optimisation**

The mentioned joint optimisation of computing, network and storage resources incurs interesting research questions, for example on how to characterise resources (e.g., finding a suitable level of granularity), which information source to leverage, how to distribute resource information, and how to arrive at a consistent view on the available system resource for enabling robust joint optimisation.

**Piccolo Protocols**

Whereas Piccolo aims at leveraging existing protocols for better supporting legacy implementations, there is significant merit in exploring new approaches, e.g., ICN-based systems, that enable new forms of splitting the work between end systems and forwarding functions.

**Decentralisation**

Whereas most distributed computing systems employ centralised resource and compute-task management, we propose an alternative decentralised approach, using an auction-based resource allocation that has promising potential for enabling optimal, fully decentralised resource allocation in heterogeneous environments.

**Security and Privacy Guarantees**

One of the motivations for edge computing systems such as Piccolo is privacy-preserving analytics (at the edge, so that raw data does not have to leave certain trust perimeters). While this is in principle a valid goal, it is challenging to actually guarantee certain levels of data retention, privacy and trust in underlying execution environments. We have described promising directions (e.g., for secure networked containers) but anticipate additional research challenges in this context.

## 5.3 Next Steps

As a next step, we are going to develop and experiment with Piccolo platforms that implement the architectural invariants (or a selection at first), for example for re-imagining the described use cases such as vision processing. The results from the experiments will then inform the refinement of our architecture and the development of specific mechanisms, for example Piccolo protocols for in-network computing and algorithms for joint resource optimisation.

# References

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

[2] Piccolo Project. *Piccolo Node Definition*. Deliverable D2.1. 2021.

[3] Bjarne Stroustrup. *The C++ programming language*. 3rd. Addison-Wesley, 1997. ISBN: 978-0-201-88954-3.

[4] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. Object Management Group, 2012. URL: `http://www.omg.org/spec/SysML/1.3/`.

[5] OMG. *Business Process Model and Notation (BPMN), Version 2.0.2*. Object Management Group, Dec. 2013. URL: `http://www.omg.org/spec/BPMN/2.0.2`.

[6] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[7] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 1973, pp. 235–245.

[8] Carl Hewitt and Henry G. Baker. "Laws for Communicating Parallel Processes". In: *Proceedings of the IFIP Congress 1977*. 1977, pp. 987–992.

[9] Carl Hewitt. "Actor Model for Discretionary, Adaptive Concurrency". In: *Computing Research Repository (CoRR)* abs/1008.1459 (2015). version 38. URL: `http://arxiv.org/abs/1008.1459v38`.

[10] Raphael Hetzel. *Decentralized Actor-based Orchestration of Networked Microcontrollers*. Master's Thesis (unpublished). Technical University of Munich. 2020.

[11] Hugo Haas and Allen Brown. *Web Services Glossary*. Accessed: 2021-03-10. 2004.

[12] Marco Andreas Wagner. "An adaptive Software and System Architecture for Driver Assistance Systems applied to truck and trailer combinations". doctoralthesis. Universität Koblenz-Landau, Universitätsbibliothek, 2015, p. 180.

[13] Google Developers. *Android OS: Guide to app architecture*. Accessed: 2021-03-10. 2021.

[14] Eclipse Foundation. *OSGi Specifications*. `http://docs.osgi.org/specification/`. Accessed: 2021-03-22. 2021.

[15] Object Management Group. *Data Distribution Service (DDS) Specification v1.4*. Tech. rep. Accessed: 2021-03-10. 2015.

[16] OPC Foundation. *Unified Architecture Part 1: Overview and Concepts*. Tech. rep. Accessed: 2021-03-10. 2017.

[17] James Lewis and Martin Fowler. *Microservices*. `https://martinfowler.com/articles/microservices.html`. Accessed: 2021-03-15.

[18] ETSI. *Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV*. 2021.

[19] CNCF. *CNCF Cloud Native Interactive Landscape*. 2021. URL: `https://landscape.cncf.io/` (visited on 03/08/2021).

[20] CNCF. *CNCF Cloud Native Interactive Landscape*. 2021. URL: `https://landscape.cncf.io/serverless` (visited on 03/08/2021).

[21] Stratis Ioannidis and Edmund Yeh. "Jointly Optimal Routing and Caching for Arbitrary Network Topologies". In: *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ICN '17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 77–87. ISBN: 9781450351225. DOI: `10.1145/3125719.3125730`. URL: `https://doi.org/10.1145/3125719.3125730`.

[22] Nitinder Mohan et al. "Pruning Edge Research with Latency Shears". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 182–189.

[23] Mahadev Satyanarayanan. "The emergence of edge computing". In: *Computer* 50.1 (2017), pp. 30–39.

[24] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. "Can Homomorphic Encryption Be Practical?" In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 113–124. ISBN: 9781450310048. DOI: `10.1145/2046660.2046682`. URL: `https://doi.org/10.1145/2046660.2046682`.

[25] X. Liu et al. "Privacy-Preserving Outsourced Calculation Toolkit in the Cloud". In: *IEEE Transactions on Dependable and Secure Computing* 17.5 (2020), pp. 898–911. DOI: `10.1109/TDSC.2018.2816656`.

[26] *Intel SGX and side-channels*. `https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html`. Accessed: 2021-02-26.

[27] *Linux Containers*. `https://stgraber.org/2014/01/17/lxc-1-0-unprivileged-containers/`. Accessed: 2021-02-26.

[28] *Docker*. `https://www.docker.com/`. Accessed: 2021-02-26.

[29] *Solaris Solaris Containers*. `https://docs.oracle.com/cd/E36784_01/html/E36848/zones.intro-1.html`. Accessed: 2021-03-23.

[30] Poul-Henning Kamp, Robert Watson, and rwatson@freebsd Org. "Jails: Confining the omnipotent root". In: (Mar. 2021).

[31] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework?" In: *Queue* 11.11 (Dec. 2013), pp. 30–44. ISSN: 1542-7730. DOI: `10.1145/2557963.2566628`. URL: `https://doi.org/10.1145/2557963.2566628`.

[32] G. Coker et al. "Principles of remote attestation". In: *International Journal of Information Security* 10 (2011), pp. 63–81.

[33] Paul England et al. *RIoT - A Foundation for Trust in the Internet of Things*. Tech. rep. MSR-TR-2016-18. Apr. 2016. URL: `https://www.microsoft.com/en-us/research/publication/riot-a-foundation-for-trust-in-the-internet-of-things/`.

[34] Vinnie Scarlata et al. “Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives”. In: 2018.

[35] Victor Costan, Ilia Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[36] Dirk Kutscher, Teemu Karkkainen, and Joerg Ott. *Directions for Computing in the Network*. Internet-Draft draft-kutscher-coinrg-dir-02. Work in Progress. Internet Engineering Task Force, July 2020. 22 pp. URL: https://datatracker.ietf.org/doc/html/draft-kutscher-coinrg-dir-02.

[37] D. L. Tennenhouse and D. J. Wetherall. “Towards an active network architecture”. In: *Proceedings DARPA Active Networks Conference and Exposition*. May 2002, pp. 2–15. DOI: 10.1109/DANCE.2002.1003480.

[38] T. v. Eicken et al. “Active Messages: A Mechanism for Integrated Communication and Computation”. In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 1992, pp. 256–266. DOI: 10.1109/ISCA.1992.753322.

[39] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/moritz.

[40] Irene Zhang et al. “Customizable and Extensible Deployment for Mobile/Cloud Applications”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 97–112. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang.

[41] Pat Bosshart et al. “P4: Programming Protocol-Independent Packet Processors”. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[42] The P4.org Architecture Working Group. *P416 Portable Switch Architecture (PSA)*. URL: https://p4.org/p4-spec/docs/PSA.html (visited on 03/22/2021).

[43] The P4.org Architecture Working Group. *P4Runtime Specification – v1.0.0*. 2019. URL: https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html (visited on 03/22/2021).

[44] Frederik Hauser et al. *A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research*. 2021. arXiv: 2101.10632 [cs.NI].

[45] Rui Miao et al. “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 15–28. ISBN: 9781450346535. DOI: 10.1145/3098822.3098824. URL: https://doi.org/10.1145/3098822.3098824.

[46] X. Chen et al. “P4SC: Towards High-Performance Service Function Chain Implementation on the P4-Capable Device”. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2019, pp. 1–9.

[47] R. Datta et al. “P4Guard: Designing P4 Based Firewall”. In: *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*. 2018, pp. 1–6. DOI: 10.1109/MILCOM.2018.8599726.

[48] Roshan Sedar et al. “Supporting Emerging Applications With Low-Latency Failover in P4”. In: *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*. NEAT ’18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 52–57. ISBN: 9781450359078. DOI: 10.1145/3229574.3229580. URL: https://doi.org/10.1145/3229574.3229580.

[49] F. Cugini et al. “P4 In-Band Telemetry (INT) for Latency-Aware VNF in Metro Networks”. In: *2019 Optical Fiber Communications Conference and Exhibition (OFC)*. 2019, pp. 1–3.

[50] *Spark Overview*. https://spark.apache.org/docs/latest/. Accessed: 2021-03-10.

[51] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[52] *What is Apache Flink? – Architecture*. https://flink.apache.org/flink-architecture.html. Accessed: 2021-03-01.

[53] Paris Carbone et al. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Engineering Bulletin* 38 (Jan. 2015).

[54] *Flink Architecture*. https://ci.apache.org/projects/flink/flink-docs-stable/concepts/flink-architecture.html. Accessed: 2021-03-01.

[55] *A Deep Dive into Rescalable State in Apache Flink*. https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html. Accessed: 2021-03-01.

[56] *Data Streaming Fault Tolerance*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/stream_checkpointing.html. Accessed: 2021-01-01.

[57] *AKKA*. https://akka.io/. Accessed: 2021-03-09.

[58] *A Deep-Dive into Flink’s Network Stack*. https://flink.apache.org/2019/06/05/flink-network-stack.html. Accessed: 2021-01-01.

[59] Jay Kreps et al. *Kafka: a distributed messaging system for log processing. NetDB’11*.

[60] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide*. 1st ed. O’Reilly Media, July 2017. ISBN: 978-1-491-93616-0.

[61] Patrick Hunt et al. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010. URL: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems.

[62] *DOCUMENTATION*. https://kafka.apache.org/0100/documentation.html. Accessed: 2021-03-04.

[63] International Electrotechnical Commission. *IEC 61499 Function blocks - Parts 1-4*. 2012.

[64] International Electrotechnical Commission. *IEC 61131 Programmable Controllers - Parts 1-10*. 2003.

[65] International Electrotechnical Commission. *IEC 61131-3 Programmable Controllers Part3: Programming languages*. 2009.

[66] Christopher Meiklejohn and Peter Van Roy. “Lasp: a language for distributed, eventually consistent computations with CRDTs”. In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. 2015, pp. 1–4.

[67] Marc Shapiro et al. “A comprehensive study of convergent and commutative replicated data types”. In: (2011).

[68] *LASP*. https://github.com/lasp-lang/lasp. Accessed: 2021-02-26.

[69] Michał Król et al. “Compute First Networking: Distributed Computing Meets ICN”. In: *Proceedings of the 6th ACM Conference on Information-Centric Networking*. ICN ’19. Macao, China: Association for Computing Machinery, 2019, pp. 67–77. ISBN: 9781450369701. DOI: 10.1145/3357150.3357395. URL: https://doi.org/10.1145/3357150.3357395.

[70] Manolis Sifalakis et al. “An Information Centric Network for Computing the Distribution of Computations”. In: *Proceedings of the 1st ACM Conference on Information-Centric Networking*. ACM-ICN ’14. Paris, France: Association for Computing Machinery, 2014, pp. 137–146. ISBN: 9781450332064. DOI: 10.1145/2660129.2660150. URL: https://doi.org/10.1145/2660129.2660150.

[71] Michał Król et al. “RICE: Remote Method Invocation in ICN”. In: *Proceedings of the 5th ACM Conference on Information-Centric Networking*. ICN ’18. Boston, Massachusetts: Association for Computing Machinery, 2018, pp. 1–11. ISBN: 9781450359597. DOI: 10.1145/3267955.3267956. URL: https://doi.org/10.1145/3267955.3267956.

[72] M. Zhang, V. Lehman, and L. Wang. “Scalable name-based data synchronization for named data networking”. In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057193.

[73] *Erlang OTP Design Principles - 9 Distributed Applications*. https://erlang.org/doc/design_principles/distributed_applications.html. Accessed: 2021-03-11.

[74] 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.

[75] Waheed Iqbal et al. “Adaptive resource provisioning for read intensive multi-tier applications in the cloud”. In: *Future Generation Computer Systems* 27.6 (2011), pp. 871–879. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2010.10.016. URL: https://www.sciencedirect.com/science/article/pii/S0167739X10002098.

[76] W. Shi et al. “An Online Auction Framework for Dynamic Resource Provisioning in Cloud Computing”. In: *IEEE/ACM Transactions on Networking* 24.4 (2016), pp. 2060–2073. DOI: 10.1109/TNET.2015.2444657.

[77] Tommy Andersson and Albin Erlanson. “Multi-item Vickrey–English–Dutch auctions”. In: *Games and Economic Behavior* 81.C (2013), pp. 116–129. DOI: 10.1016/j.geb.2013.05.001. URL: https://ideas.repec.org/a/eee/gamebe/v81y2013icp116-129.html.

[78] L. S. Shapley and M. Shubik. “The assignment game I: The core”. In: *International Journal of Game Theory* 1.1 (Dec. 1971), pp. 111–130. ISSN: 1432-1270. DOI: 10.1007/BF01753437. URL: https://doi.org/10.1007/BF01753437.

[79] *Docker Swarm*. https://docs.docker.com/engine/swarm/. Accessed: 2021-02-26.

[80] *Kubernetes*. https://kubernetes.io/. Accessed: 2021-02-26.

[81] Eric A. Brewer. “Kubernetes and the Path to Cloud Native”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC ’15. Kohala Coast, Hawaii: Association for Computing Machinery, 2015, p. 167. ISBN: 9781450336512. DOI: 10.1145/2806777.2809955. URL: https://doi.org/10.1145/2806777.2809955.

[82] 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.

[83] D. Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *Computer* 45.2 (2012), pp. 37–42. DOI: 10.1109/MC.2012.33.

[84] D. Box et al. *Simple object access protocol (SOAP) 1.1*. Jan. 2000.

[85] Khashayar Kamran, Edmund Yeh, and Qian Ma. “DECO: Joint Computation, Caching and Forwarding in Data-Centric Computing Networks”. In: *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing*. Mobihoc ’19. Catania, Italy: Association for Computing Machinery, 2019, 111â120. ISBN: 9781450367646. DOI: 10.1145/3323679.3326509. URL: https://doi.org/10.1145/3323679.3326509.