PICCOLO

CELTIC-NEXT

# Deliverable D3.2

# Initial Design of the Piccolo infrastructure

| | |
|---|---|
| Editor: | Philip Eardley – BT |
| Deliverable nature: | Report (R) |
| Dissemination level: (Confidentiality) | Public |
| Contractual delivery date: | September 30th, 2021 |
| Actual delivery date: | September 30th, 2021 |
| Suggested readers: | Researchers, developers and technologists interested in edge computing and the convergence of networking and computing. |
| Version: | 1 |
| Total number of pages: | 57 |
| Keywords: | Architecture, Distributed Computing, Protocols |


***Abstract***

This document describes initial designs for the Piccolo system architecture, in particular orchestration and resource management aspects.

## 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] Task 3.1, 3.2, 3.3 and 3.4
[Document title] D3.2 Initial Design of the Piccolo Infrastructure
[Editor: Name, company] Philip Eardley, BT
[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.

This document presents our work about the overall distributed computing 'infrastructure', at the half-way stage of the project's two year timespan.

We have focused our work on advancing various implementations in order to get the Proof-of-Concept demonstrators working as soon as possible. As a side effect, it shows that the Piccolo mechanisms and APIs can be implemented on a variety of platforms and programming environments. This bottom-up work complements our earlier document, which took more of a top-down approach (Del3.1, Architectural invariants for distributed computing and Technical requirements).

The latest progress is reported about:

- **Overall architecture design**, with a particular focus on the critical topic of identification and addressing.
- **Orchestration elements** - we explore several approaches in different contexts, including hierarchical, decentralised and semi-decentralised orchestration.
- **Resource management elements** - again, we explore several approaches in different contexts, including decentralised, auction-based and reservation-based resource management.
- Other elements - which covers some other topics that we have investigated.

 

# List of Authors


| Company | Author |
|---|---|
| ARM Ltd. | Chris Adeniyi-Jones |
| British Telecommunications plc | Adam Broadbent, Philip Eardley, Andy Reid, Kostas Antoniou, Peter Willis |
| Fluentic Networks Ltd. | Ioannis Psaras, Alex Tsakrilis |
| InnoRoute GmbH | Andreas Foglar, Marian Ulbricht, Surik Krdoyan |
| 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, Nitinder Mohan |
| University of Applied Science Emden/Leer | Dirk Kutscher, Laura al Wardani, T M Rayhan Gias |

# Table of Contents



 

 

# List of Figures



 

# Abbreviations

**ACK** ACKnowledgement
**AIMD** additive-increase/multiplicative-decrease
**API** Application Programming Interface
**AppSP** Application/Service Provider
**APU** Accelerated Processing Unit
**CAM** Content Addressable Memory
**CDN** Content Delivery Network
**CRDT** Conflict-free replicated data type
**CRDT** Conflict-free Replicated Data Type
**CV** Continuous Verification
**CV-OAM** Continuous Verification - Operations, Administration and Maintenance
**EE** Execution Environment
**EWBI** East West Bound Interface
**FaaS** Function as a Service
**FLOPS** Floating point operations per second
**FPGA** Field Programmable Gate Array
**GPU** Graphics Processing Unit
**GSMA** GSM Association
**ICN** Information-Centric Networking
**IGP** Interior Gateway Protocol
**INCP** In-network Computing Provider
**IoT** Internet of Things
**ISP** Internet Service Provider
**IT** Information Technology
**MIPS** Mega Instructions Per Second
**MQTT** Message Queuing Telemetry Transport
**NACK** Not ACKnowledgement
**NAT** Network Address Translation
**NBI** NorthBound Interface
**NDN** Named Data Networking

| | |
|---|---|
| **NFaaS** | Named FaaS |
| **NFN** | Named Function Networking |
| **OS** | Operating System |
| **OTP** | Open Telecom Platform |
| **PID** | Process ID |
| **PoC** | Proof of Concept |
| **QoS** | Quality of Service |
| **REST** | Representational State Transfer |
| **RPC** | Remote Procedure Call |
| **SCONE** | Secure CONtainer Environment |
| **SDH** | Synchronous Digital Hierarchy |
| **SDN** | Software-Defined Networking |
| **SLA** | Service Level Agreement |
| **SLA** | Service-Level Agreement |
| **TEE** | Trusted Execution Environment |
| **TINC** | Trusted In-Network Computing |
| **TSN** | Time-Sensitive Networking |
| **VED** | Vickrey-English-Dutch |
| **VM** | Virtual Machine |
| **YAML** | Yet Another Markup Language |

# 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 Application Programming Interfaces (APIs), enables efficient resource allocation, availability and scalability support, and other services.

Piccolo is now half-way through its two year timespan. This document presents our work about the overall distributed computing 'infrastructure' ; the companion document Del2.2 [1] covers the individual nodes ; whilst Del1.2 [2] reports on the Proof of Concept demos. Therefore the document provides a snapshot.

In the earlier deliverable, Del3.1 [3], we studied technical requirements and design goals that arise from several interesting use cases, and derived some 'architectural invariants' - architectural pillars and system properties.

In this document we advance the work in various practical implementations, concentrating on two main topics: orchestration and resource management - with sub-sections about the detailed work for each proof of concept demonstrator. At this point we have taken a bottom-up approach, in order to get the demonstrators working. It also shows that the Piccolo mechanisms and APIs can be implemented on a variety of platforms and programming environments. Next, planned work will re-integrate into a more conceptually unified framework.

The document is structured as follows:

- Architecture invariants.
- **Overall architecture design**, with a particular focus on the critical topic of identification and addressing.
- **Orchestration elements** - a hierarchical orchestrator ; a semi-decentralised orchestrator ; a decentralised orchestrator for an ICN-based dataflow system ; a hybrid approach ; orchestration for the Smart factory use case and node type.
- **Resource management elements** - an ICN-based dataflow system, with decentralised scaling ; an auction-based marketplace approach to resource allocation ; and resource management for time syncronised networking.
- Other elements - a description of the in-network compute simulator ; the concept of a federation of operators that all offer in-network compute in a consistent manner ; and an investigation into whether Erlang can be suitable for implementing a Continuous Verification - Operations, Administration and Maintenance (CV-OAM) environment for a network provider.

# 2 Architecture invariants

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.

In our earlier Deliverable D1.1 [4], we studied four use cases areas that are promising for in-network computing: vision processing; connected and automated driving; smart factory; and network operations and management. We analysed the requirements and design goals that arise from these use cases, which fall into the following categories:

- Heterogeneity: a Piccolo edge compute system is likely to include multiple sorts of hardware (such as Graphics Processing Unit (GPU) as well as general purpose processors), operating system, virtualisation solution (virtual machine, container or bare metal) and communications models (including data stream processing, publish /subscribe and remote method invocation).

- Initialisation capabilities: the Piccolo system needs to support discovery of the preferred compute node and resource management. The Piccolo functions (the compute tasks) need to be placed sensibly - on a single node, or more likely distributed across several nodes. The placements needs to bear in mind the end usersâ requirements and the operator's policies.

- In-life capabilities: for example, resilience and scaling, so that applications can scale and continue to work well despite failures or bottlenecks. The system also needs to handle mobility of the end hosts, whilst continuing to distribute compute tasks efficiently in the network.

- Security and privacy: to authorise compute inputs, secure the execution of compute tasks and ensure that the compute results are correct, valid, verified and can be trusted. These features must work with multi-tenancy, since there will be several independent tenants (customers) being served by the same Piccolo system.

In the context of the above requirements, the project has derived some 'invariants' - architectural pillars and system properties - that are summarised below:

- **Piccolo agent** - this entity runs on each Piccolo node. It manages the execution environment of the node (which provides the runtime environment that Piccolo functions are executed within) and also provides the control plane API, which interfaces the node and its capabilities to the other nodes in the Piccolo system.

- **Actor model as a fundamental abstraction** - All computations are modelled using 'actors': computational entities that only communicate using messaging and maintain local, mutable state that is not accessible by other actors. When an actor processes a message, it may send messages to other actors (asynchronously), modify its local state, and create additional actors. The Actor approach is followed by the Erlang and Akka runtime systems, for instance. For Piccolo, Actors provide a natural abstraction for state and lifecycle rules, network transparency, isolation, scaling and resilience.

 

- **Location-independent, nested naming** - in Piccolo applications should use names, so that they are address agnostic. Actors (functions) are named independently from nodes (infrastructure), so that names can be nested, and in fact there can be multiple layers of nesting ('recursion'). Hence calling a function (which means sending a message to an actor) can result in a chain of additional function (actor) invocations. Another example of nesting is a function that runs on a container, and the container is actually in a Virtual Machine (VM), which in turn runs on bare metal. The namespaces at the different levels are totally independent, so that name changes aren't required at a 'client' layer when there's a change in the underlying infrastructure (including an actor moving to a different node).

- **Dynamic loading of functions onto nodes** - functions can be instantiated on nodes dynamically. One purpose is clearly to dynamically execute an application or service. Linking to the previous invariant, it can also be part of achieving scalability and availability (resilience), and handling mobility.

- **Joint optimisation** - Piccolo seeks to make a decision (at run-time) with respect to allocation of compute, networking and storage resources (and in principle other resources such as energy), in combination with the overall compute graph construction. We would like to 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 and policies. Approaches to explore include leveraging Interior Gateway Protocol (IGP) information and auction-based resource allocation.

- **Secure virtual processing** - most applications will require a secure, multi-tenant processing environment, with isolation guarantees so one tenant cannot interfere with the operation of another tenant. The protection must extend from application code (including an execution environment) to application data. Examples of secure compute are Secure CONtainer Environment (SCONE), a secure container framework, and Arm's confidential compute architecture.

- **Reliability through a generalised end-to-end principle** - the end points of the distributed function must implement reliability and consistency measures themselves, and cannot regard reliability and consistency as something which can be delegated to capabilities within the infrastructure. It also implies that intermediate components do not need to be designed to guarantee reliability or consistency and can be optimised for latency and availability. This approach is analogous to the end-to-end principle of 'soft state' held by Internet protocols such as TCP, PPP, and many routing protocols.

- **Support heterogeneity of types of nodes and links** - in order to allow the most adequate platform and communication technologies to be chosen for specific functions in a distributed system, Piccolo is agnostic to the hosting platform and supports different interaction types.

The architectural invariants are described in more detail in Deliverables D2.1 [5] and D3.1 [3], respectively about the individual Piccolo node and the distributed system.

In this document we progress our architectural work, developing the architectural invariants into a more detailed initial design for the Piccolo infrastructure, whilst aspects relating to a Piccolo node are refined in the companion document, D2.2 [1].

# 3 Overall architecture design

In this section, we outline our architectural design tenets, following from and complementing the invariants discussed in the previous section. Any architectural design needs to position itself with respect to other technologies in a given space, embracing, replacing, or otherwise interacting with them. This also holds for the Piccolo architecture and in-network as well as edge computing. With the Internet as the context and distributed service provisioning as the goal, the key – and essential! – invariant guiding the overall architectural design is supporting heterogeneity and, to this end, we choose to position the Piccolo architecture in a way able to embrace and interact with other systems, rather than prescribing a particular encompassing system design in detail. This becomes manifest in our design principles.

Heterogeneity is key because of the wide variety of infrastructure – devices with their associated operating systems and networks – both the variety of services to be realized for different use cases and the frameworks and programming languages ultimately used to realize those use cases. For the Piccolo architecture to be of broad use, its design must respect these differences - and even more dimensions of heterogeneity as we will see below. This also applies to the varied environments in which Piccolo systems will be deployed: we obviously have an extensive Internet ecosystem, in which cloud and in-network, edge, and Content Delivery Network (CDN) resources are complemented by user devices, on-premise equipment, and possibly further sensing, actuation, interaction, and computation infrastructure from third parties. An in-network compute architecture needs to get along with and embrace and thus be able to draw resources from all of the above. That is, it is imperative that the Piccolo architecture be designed to be part of something else! While Piccolo has a long-term vision of network architectures that would support in-network computing better, its architectural design will be able to operate in present-day network architectures, including the Internet, information-centric networks, and domain-specific networks. The same holds for node architectures, processing flows and communication demands of applications, etc.

These observations directly lead to the consequence of a fairly minimal design. In history, the Internet Protocol has succeeded by placing minimal requirements on the underlying networks, allowing them to be easily embraced as part of a larger vision, thus not placing undue burden and complexity on the existing networks. In Piccolo, we follow a similar approach: we place only minimal requirements on nodes to be part of the Piccolo architecture. We do not require a specific operating system, abstraction level, virtualization technology, language, etc. but only ask systems to offer two minimal interfaces towards a Piccolo agent:

- a protocol for the exchange of information with other Piccolo nodes
- local APIs for the Piccolo agent to invoke and interact with execution environments and actors (see Del2.2 [1]).

For both interfaces, the semantics need to be agreed upon but the realization may be platform-specific in the case of APIs and domain- or deployment-specific for the protocols. For open deployments, a “standardized” protocol specification is required.

 

This minimal design leaves most other aspects up to the specific applications: for example, which protocols actors use to exchange information, what their communication patterns and performance demands are, their scale and topological organization, among others. The Piccolo architecture only facilitates finding suitable nodes to run such actors, instantiate them, and then provide an environment in which the actors can exchange information with their peers according to their needs. But Piccolo stays clear of the actual information exchange, even though it could offer simple message passing to facilitate the interaction.

In this section, we first recap the architectural elements of Piccolo in subsection 3.1. We then discuss the key concepts of node identification and addressing in subsection 3.2. We finally turn our attention to orchestration and resource management principles, the key constituents of the architecture in section 3.3, which we will detail further in later chapters.

## 3.1 Architecture Outline

The Piccolo architecture is comprised of node execution environments (nodes with agents that enable execution of actor code and communication with other actors), a fundamental message-based communication service for actors (that enables additional interaction styles), and an infrastructure resource management framework, where agents on nodes perform management functions on behalf of the instantiated actors on these nodes.

Agents provide APIs for inter-actor communication and management functions to actors and can implement them in specific ways (depending on the Piccolo context), i.e., there can be different implementations of Piccolo systems that adhere to this conceptual framework but use specific distributed computing environments and protocols to implement them.

Two examples would be: an Information-Centric Networking (ICN)-based Piccolo system that implements all communication with Named Data Networking protocols; and an Erlang-based system that leverages Erlang Distribution (the already existing distributed computing service in Erlang). In both environments, actors would use isomorphic APIs, but the details of working with the API as well as the resulting agent actions would be specific to the respective environment.

One important aspect in Piccolo's distributed computing system is resource management. This includes *allocation of computing resources as well as communication resources* (network capacity). *Specific instantiations of the Piccolo concepts* can provide their own resource model and specific ways for describing and managing these resources. In the current state of the project, we are experimenting with different environment-specific approaches, and the insights from these experiments will lead to a more general model and possibly to corresponding API updates.

Running Piccolo systems can be supported by a bespoke orchestration system that monitors availability, performance and that can assist in instantiating applications, managing their initial and run-time resource consumption etc. However, there are some Piccolo systems that do not require an explicit, central orchestration service, such as the ICN Dataflow system introduced in section 4.3.2. In such a

 

system, the respective agents would implement resource-management-related functions (e.g., scaling out) in a distributed manner, i.e., employing decentralized orchestration.

One particularly important concept in the Piccolo architecture is *identification* and *naming*, pertaining to how to refer to underlying infrastructure, actors in a distributed system, computation results etc. We will discuss this concept in more detail in the next section.

## 3.2 Piccolo Approach to Identification and Addressing

As Piccolo is creating an extensible distributed framework, consistent identification and addressing *of actors across potentially multiple nodes* is essential to achieving this objective. There are two aspects to this identification: (1) the names and identifiers used within a particular Piccolo service (instance), i.e., the name space from which actor names and identifiers are chosen. This namespace is relevant to the composition of services (or: applications) from actors and visible and relevant to the internal service logic of each individual service. This name space is shielded from the surrounding environments, and the service components, i.e., actors, have exclusive use of their own name space and identifiers. (2) The Piccolo nodes on which the Piccolo actors are executed "live" in their surrounding networking environment which defines the name space for the names and identifiers that are used to exchange data between the nodes in that networking environment (e.g., using IPv4, IPv6, NDN, L2 networks). This name space is under control of the surrounding network. It is up to the Piccolo orchestration and management to provide adequate mappings between these address spaces, which may take many different shapes, including (but not limited to) name/address resolution, identifier mapping, creating overlays, tunneling, and/or indirect message forwarding via Piccolo agents. In this section, we discuss the architectural aspects of naming in Piccolo.

As with many aspect of Piccolo architecture, at this stage of the project, the emphasis is on developing proof of concepts for "bottom-up" use cases rather than "top-down" architecture. Developing top-down architecture will follow over the coming months by developing common abstractions from the Proof of Concepts (PoCs). This approach ensures that the common architecture is suitable for and tested against a wide variety of scenarios.

Identification and addressing schemes fit within this framework for the project. Each PoC currently exploits identification and addressing suitable for its use case without imposing top-down restrictions at this stage.

However, identification and addressing is sufficiently important that it is helpful to have formed a initial view on how the many different schemes from the PoCs might be harmonised. Broadly, there are two alternatives:

- Design a single consistent scheme and retro-fit the existing PoC designs to this new scheme.
- Design an architecture which can aggregate and integrate different independent schemes into a single overall architecturally consistent framework.

 

In Piccolo, we have chosen the latter option.

Piccolo's scope therefore includes many existing and well established identification and addressing schemes, for example IP addressing, the domain name system, and composite schemes like URLs, and in order for Piccolo to have a credible evolutionary path for initial deployments, it is important that these existing schemes can be included into a consistent Piccolo identification architecture.

The identification and addressing architecture in Piccolo is based a number of principles. At this stage, these principles are at a 'strawman' stage and will be developed in the next stage of the project. So some or all of them may well be refined.

- The Piccolo identification and addressing architecture is able to incorporate existing schemes as needed.
- Where possible and within the limits of existing schemes, Piccolo identification and addressing are easily extensible.
- There is great importance to the linkage between identification schemes, for example using distributed lookup systems, and there is no need to select one scheme above others.
- The architecture of identification and addressing in Piccolo is based on the dynamic construction of identifiers appropriate to a specific context/scope and the requirement is that they are unambiguous in their context.
- A specific context/scope itself has an identifier.
- A context/scope can be dynamically nested giving dynamically concatenated identifiers.
- The scheme used at one level of nesting does not need to be the same as at another level so it is possible to dynamically create new composite schemes by dynamically prefixing or suffixing one scheme with a different scheme.
- The order of concatenation is dynamic and flexible.
- While top-down global allocation schemes designed to assure global uniqueness will guarantee unambiguity in context, they can create many other restrictions which are incompatible with many of the principles set out above.
- Global uniqueness can be created dynamically if needed by dynamically placing a context within a wider context which guarantees global uniqueness.

This approach differs in emphasis to many existing schemes (including many of the systems that Piccolo includes within its architecture) which are often based on the idea of a single consistent scheme. Moreover, often with other projects the emphasis is on guaranteeing global uniqueness within an identification scheme and another related focus is often fully internalising and eliding any linkage to any different internal identification scheme, for example making invisible the specific physical

 

location of a function with a visible logical identity.

By way of an important illustration of the need for a flexible system is the limitation that is currently imposed by the URL scheme used in micro-services. In the URL, the domain name/IP address must precede the name of the function being called which means that the location of the function must be known and defined at the time of the call. This is not aligned with a basic objective of Piccolo where the function can be called by name and the call is forwarded transparently to the most efficient location hosting the function. This issue is one of the issues addressed by service mesh solutions.

## 3.3 Orchestration and Resource Management

As discussed in the previous section, a namespace forms the universe in which Piccolo actors exchange information to realize an application or service. Once active, the actors run on interconnected Piccolo nodes. Before that, these nodes need to be chosen, the actor code moved to and instantiated on them, and the necessary network mappings need to be put in place. Exercising these parts is the task of the Piccolo orchestration system and, for per-node execution, that of the individual Piccolo agents running on these nodes. Specifically, two tasks make up the overall Piccolo orchestration:

Firstly, this requires the Piccolo orchestration to understand which Piccolo nodes are actually capable of executing the respective actors. This is limited by the hardware platform, instruction set architecture, operating system, and virtualization environment (and thus the execution environment) available on each Piccolo node: only nodes compatible with the actors needs can be chosen. Moreover, actors may require specific peripheral devices, e.g., sensors or actuators, or GPUs, for their operation: only nodes with such peripherals can be chosen. Finally, Piccolo nodes may have vastly different capabilities, in terms of main memory, computing power, storage capacity, and network interfaces, and they may face further limitations such as running on battery. A meta description can be used to express, on the one hand, the properties, capabilities, and constraints of each Piccolo node and, on the other hand, the requirements of the individual actors (and possibly the changes in demand as they scale). This process yields a set of nodes suitable to execute the application actors.

Secondly, the Piccolo orchestration is responsible for determining how to distribute the actors of an application to one or more of these suitable Piccolo nodes – and then invoke the actual distribution and/or instantiation of the actor code via the per-node Piccolo agents. This means that in-network resources – computation and storage and well as messaging capacities across the links and paths interconnecting the chosen Piccolo nodes – have to be allocated to the respective application for the expected time of its execution (cf. joint optimization). For this purpose, the Piccolo orchestration needs to be aware of the available (and possibly the total) resources on the Piccolo nodes that should be considered for execution. Since Piccolo aims at short application response times, actors should be distributed to nodes close to each other and close to their user(s); the network capacities and especially one-way delays between nodes may thus guide which subset of nodes is suitable to run an application instance (for a given set of users). To this end, metadata about the interactions among actors (frequency, latency demands, data volume per transaction) could describe the placement con-

 

straints to be imposed on the node choice. At the same time, for a dynamic system, the overhead for (re)instantiating and moving code and/or data between nodes needs to be considered. This matching step may also involve suspending, moving, and reinstantiating other actors to make room for the actors of a new application (instance).

This orchestration and resource allocation process can be realized in many different ways. As discussed in D3.1 [3], cloud infrastructure uses a tight coupling between a central controller and the managed nodes, whereas edge systems reduce the tightness of the control loop. Fundamentally, orchestration may be centralized around a core orchestrator that takes all decisions or it may be distributed. Distributed orchestration may employ a strict hierarchy with strictly defined information and decision flows, or may be fully decentralized with equal peers negotiating which actors should run where. This may be aided by auctioning mechanisms to include the application's view and the cost in a heterogeneous environment with multiple suppliers, or it may be a single provider that could strive, e.g., for its own resource optimization. In an extreme case, the orchestration may be fully decentralized across the – then unmanaged, at least from the perspective of orchestration – nodes, realizing bottom up orchestration. In such a case, for example, an application might be brought to a compute node that determines which actors it can execute locally and which ones it needs to outsource; for the latter, the node would then contact neighboring nodes it is aware of to get help in running the actors. Status information exchange may flow in all of the above cases, from compute nodes to orchestrator nodes, among orchestrator nodes, and among compute nodes. The level of detail and the frequency of information exchange – i.e., the degree of coupling the nodes – may vary. Even within the same system, close-by nodes may exchange more detailed information more frequently compared to remote nodes, and lower hierarchy levels may perform richer information exchange compared to higher ones, etc.

The Piccolo architecture does not demand a particular mode of operation. On the Piccolo nodes, the Piccolo agent is responsible for its integration in the broader Piccolo system, which is done via its protocol interface. This interface supports exporting the (static) node properties as well as the (dynamic) resource availability. It may be used to receive instructions of what to execute (and obtain the corresponding code). But a Piccolo agent may also take local decisions, inform other nodes about its operational state (running actors, etc.) and actively ask other Piccolo nodes if they can execute one or more actors, which those may acknowledge or deny. This conceptual combination of status sharing on the one hand and sending/receiving inquiries and instruction on the other allows for the aforementioned flexibility when creating different types of Piccolo orchestrators.

In the following two chapters, we will first discuss sample Piccolo orchestrators that represent different points in the design space and then discuss Piccolo resource management mechanisms.

 

# 4 Orchestration Elements

In the past years, several different architectural solutions have been proposed to simplify the development and deployment of application and services at the network edge. Mostly inspired and driven by the design of data centre solutions, they are often found ineffective for heterogeneous deployments (including compute and network) due to their strong assumption on the underlying infrastructure. The following section present solutions to orchestrate applications and service at the network edge of heterogeneous deployments.

## 4.1 Hierarchical Orchestration over Edge Infrastructures - EdgeIO

This section presents EdgeIO – a hierarchical orchestration framework designed to support the inherent heterogeneity of hardware, software and management commonly existing in edge infrastructures. The framework enables multiple network operators to contribute their resources towards a shared infrastructure, thereby significantly reducing required investments to achieve a dense computing fabric at the edge. Simultaneously, EdgeIO employs a *delegated scheduling* mechanism that decentralizes the task placement problem across the hierarchy to support optimal application service deployment at scale. The following section discusses the general problem of scheduling and orchestration at the edge and why existing cloud-based approaches (and derived solutions) are ineffective for edge infrastructures. Further we present how the proposed hierarchical (and clustered) design of EdgeIO enables both edge resource providers to contribute and manage their resources; and developers to find optimal resources at the edge to match their service's strict operational requirements with relative ease. EdgeIO follows the heterogeneity principles core to the Piccolo operation as it enables services virtualized in variety of computational abstractions (containers, actors, functions, etc.) to operate over variety of hardware configurations.

### 4.1.1 Problem Statement

Several different understandings of edge architectures co-exist currently, all aiming to incorporate resources deployed in varying environments (home, factory, city) with varying hardware capabilities (Accelerated Processing Unit (APU), accelerators, Field Programmable Gate Arrays (FPGAs)) and behind different administrative boundaries (Internet Service Provider (ISP), industrial, crowd sourced) [6, 7]. As a result, typical software solutions for managing and deploying services at the edge, e.g. Kubernetes[1], ioFog[2], FogO5[3], KubeEdge[4], etc. are often found to be ineffective since their inherent design is often a variant of existing data centre-centric frameworks. Specifically, existing frameworks make strong assumptions about the underlying infrastructure for optimal operation,

[1] https://kubernetes.io/
[2] https://iofog.org/
[3] https://fog05.io/
[4] https://kubeedge.io/en/

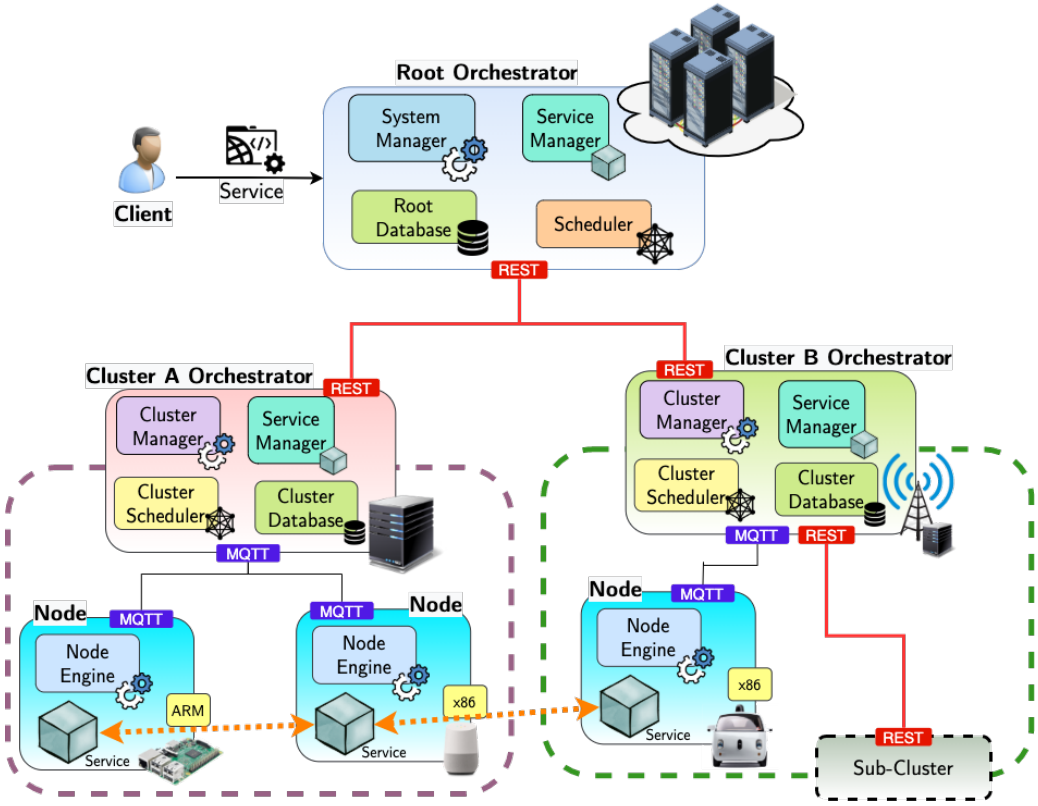


Figure 1: EdgeIO framework.

which was found to be a limiting factor when porting to edge infrastructures [8]. For example, most frameworks requires all processing resources to be in the same cluster (and therefore directly reachable from each other), just like data centres, which does not always hold true for processing units in the edge. The most inhibiting factor affecting the performance of existing orchestration frameworks in edge environments is: (i) ability to handle the heterogeneity of the devices making up the infrastructure (ii) scale of devices that span large geographical regions and (iii) different management entities that may collaborate together to contribute their resources to a larger infrastructure.

### 4.1.2 EdgeIO Description

Figure 1 shows the proposed architecture of EdgeIO. The distinguishing feature of EdgeIO to existing orchestration solutions is its semi-decentralized management by fragmenting edge infrastructure into multiple federated *clusters*. The *Root Orchestrator* is a centralized control plane in the cloud that includes modules for interfacing with clients/developers (system manager) and providing aggregated management over connected clusters; storing aggregated information of connected resources (shown as *Nodes*) and deployed services in the root database, while the *Cluster Orchestrator* handles minute management of edge resources within a cluster's administrative boundary. Each cluster represents a logical separation between edge resource groups, based on location, ownership, type, etc. [9]. Furthermore, multiple abstractions within the same organization (e.g. different location groupings of ISP base stations) can be supported as nested cluster memberships organized in a tree-like hi-

erarchy. EdgeIO integrates edge resources behind firewalls, proxies and Network Address Translations (NATs) (i.e. resources with private IPs) with the infrastructure-at-large using standardized message-passing protocols and APIs (e.g. Message Queuing Telemetry Transport (MQTT), RabbitMQ, Representational State Transfer (REST)) for information sharing and federated management. Periodically, worker nodes (or Piccolo nodes) in EdgeIO updates their local cluster orchestrator of their current load, running services and other changes in configurations (e.g. location updates). By design, the cluster orchestrator withholds detailed information about individual resources to retain administrative control. The cluster orchestrator aggregates the information of its cluster resources and sends the combined metrics to the root orchestrator.

EdgeIO's unique hierarchical orchestration is designed from ground-up to enable edge computing applications. Multiple operational entities can contribute their local deployments towards a shared edge infrastructure while retaining administrative control through EdgeIO's unique multi-cluster resource management. As a result, EdgeIO provides a unique opportunity to realize a dense compute fabric at the network edge without significant investments in deployment. Developers interested in deploying their applications at the edge submit the application code and a Yet Another Markup Language (YAML)-formatted list of Service-Level Agreements (SLAs) to the orchestrator via an API to the **service manager** at the root orchestrator. The service manager notifies the **system manager** at the root of the new deployment request, which registers the service in the local database. The system manager further contacts the **root scheduler** to calculate a priority list of clusters best suited to deploy the application.

Scheduling process in EdgeIO differs from most other orchestration frameworks to support the vast scale of edge infrastructure. Instead of scheduler in root being responsible for finding the optimal placement of developer submitted service on an edge server, EdgeIO follows *delegated scheduling*. The root scheduler only finds the target cluster for executing the application by filtering out all clusters not suitable for the task, e.g., not enough resource availability, not in the target geographical region, no support for the desired virtualization, etc. Once a suitable priority list of cluster capable of supporting the application is created, the root *delegates* the service scheduling task to the **cluster scheduler**. After receiving the placement request, the cluster scheduler calculates the optimal deployment of the service on workers within the cluster. The scheduler utilizes a *best-fit* policy where the worker node that best satisfies the operational constraints defined by the developer in the SLA of the service is selected. The cluster orchestrator further contacts the selected worker node and requests to deploy the service. Since edge infrastructures can be highly dynamic and resource utilization can change within status updates, it is quite possible that the worker selected by cluster scheduler can not support the SLA requirements of the service anymore. In this case, an explicit *reject notification* is triggered by the worker node to the cluster orchestrator, which calculates next possible resource within the cluster boundary that can support the service. If none of the resources in the cluster can support the service, the cluster orchestrator sends a Not ACKnowledgement (NACK) to root orchestrator, which in turn contacts the next cluster orchestrator in the priority list. On the other hand, if the worker node accepts the deployment request, the cluster orchestrator ACKnowledgements (ACKs) the root of the deployment. Note that since scheduling logic in EdgeIO is decoupled, each infrastructure operator can fine-tune the utilization of its resources by employing different service scheduling rules that are

only valid within that cluster boundary.

With its unique hierarchical orchestration design, EdgeIO addresses the *scalabiility* of resources and services challenge targeted by Piccolo. Both large and individual resource providers can participate in the infrastructure at large by setting up their own cluster of resources over which they retain complete management control. Thanks to the clustered design and delegated scheduling mechanism, EdgeIO also addresses the *decentralization* research challenge within Piccolo – as the orchestrator can find optimal resources for operating applications at the edge without having to the solve NP-hard scheduling problems [10, 7].

## 4.2 Self-Organizing Service Orchestation - VineIO

This section presents VineIO – an autonomous self-organizing service orchestration system that enables flexible and automated orchestration of services that react to dynamic changes in the environment. Through its unique bottom-up orchestration policy, VineIO plugs in a prominent gap in edge service placement research – *dynamicity*. This section describes the core problem that exists when scheduling mobile and flexible services of next-generation applications on an edge infrastructure. Further, we present the internal design of VineIO self-orchestration principle and how Piccolo agents managed by VineIO can automatically adapt to changes in network, environment and application operations.

### 4.2.1 Problem Description

Even though EdgeIO distributes scheduling over multiple participants forming a hierarchical scheduling mechanism allowing flexible and accelerated service deployment at the edge, it is still incapable of adapting to frequent, highly-variable changes in the (user and edge infrastructure) environment. Take for example the case of offloading driving logic for automated driving vehicles described in Piccolo Deliverable 1.1 [4] Section 3.6. Considering the automated driving vehicle moves with the certain velocity in the geographical area, the logic function that supports its operation must always remain within optimal latency of the vehicle. In this, the scheduling block of an orchestration framework designed for edge infrastructures must be able to sample the current utilization of the resources and dynamically update the service placement as the SLA degrades. However, since service deployment is a well-known NP-hard problem [7], handling scheduling in any centralized orchestrator (either at root or cluster level) makes it difficult to handle dynamic changes in the environment state. In such a scenario, the latency overhead of EdgeIO's semi-decentralized scheduling mechanism would *still* be too large to keep up with the vehicle's speed. The service placement problem becomes further complicated if one considers dynamic changes in available capacity of edge resources due to simultaneous execution of other services on the infrastructure.

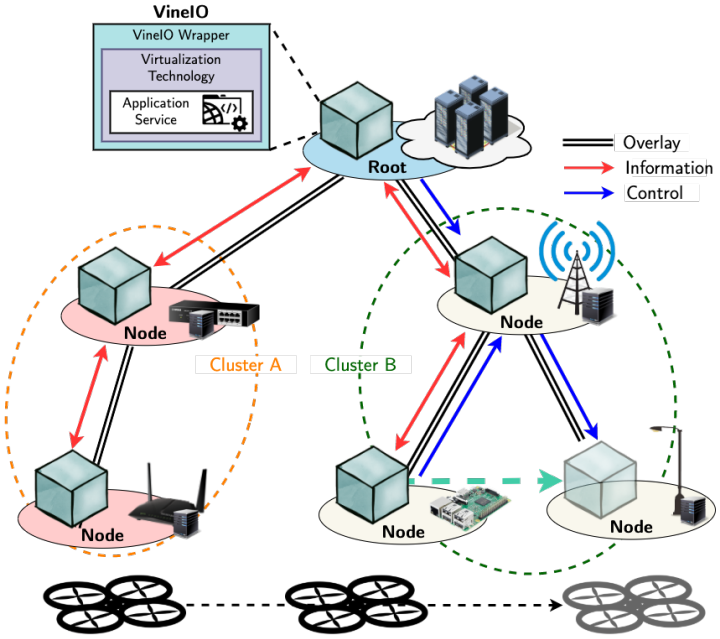


Figure 2: VineIO framework.

### 4.2.2 VineIO Description

Fig. 2 shows the proposed design and operation of *VineIO*. VineIO provides autonomous management and self-organizing capabilities to application services and will be implemented as an extension to existing virtualization technologies (e.g. virtual machine, container, etc.). Specifically, VineIO enables virtualized services to self-monitor their Quality of Service (QoS) metrics (e.g. user requests, location, hardware utilization, etc.) and trigger autonomous migration/replication to another edge server in case they do not meet the SLA targets. The core component of VineIO is a utility function [11] that is responsible for intelligently identifying (present and near-future) changes in the application performance. By default, VineIO computes the utility at a network location $j$ which serves $\lambda_i$ requests per second from a subnet $i$ as follows:

$$\mathcal{U}_j = ((1-w)b_j + w \sum_{\forall i \in S} \lambda_i l_{i,j}) \tag{1}$$

where $b_j$ is the cost of running VineIO instance at location $j$, $l_{i,j}$ is the latency experienced by users from subnet $i$ if served from network location $j$, and $w$ is the weight parameter ($0 \leq w \leq 1$) for tuning the importance of the cost or latency component. In addition to function (1), VineIO allows to plug-in other utility functions to further fine-tune the deployment of services in edge environments. As a result, VineIO allows seamless movement of service within the edge infrastructure (spanning many administrative boundaries) while relieving the application developer to "micro-manage" a large set of deployed services at global scale and allowing services to react faster to the changes in the environment.

Application deployment over VineIO is as follows: At the beginning, a single instance of the VineIO encapsulated service is deployed over some node in the infrastructure. Each VineIO instance includes

the developer submitted application service wrapped in a compatible virtualization technology (like containers) and includes high-level SLA guidelines submitted by the developer that will guide the future migration/replication decisions of the service. This instance assumes the (logical) role of the root node (see Figure 2) and is, therefore, responsible for managing further deployment. The VineIO root is also the direct point-of-access for the developer that contacts the node directly to assess the current health and deployment for the service in the infrastructure. The deployment of root instance can is possible either by the developer directly or through third-party orchestration framework like EdgeIO (see Section 4.1).

VineIO service, once successfully deployed at the edge, forms a logical tree-like overlay attaching itself as a child to VineIO manager in the root. The service tree grows organically as VineIO's replicate within the infrastructure, each replica becoming a child of its parent. The information and control flow within the VineIO overlay is hierarchically organized (similar to hybrid DNS operation) where each node directly manages and receives information from direct children. The VineIO service at the edge continually monitors its application's QoS and shares the performance metrics with its parent. In return, the parent VineIO aggregates all its children's information and transmits it to its own parent and children. If the VineIO service detects that the QoS of the encapsulated application is unlikely to meet the SLA targets, it informs the in-built cluster orchestrator to find a better-suited resource for migration/replication – prioritizing the locality constraints of edge services. The request recursively moves up the cluster hierarchy if no suitable resource is available at the cluster. Due to its unique adaptive operation, VineIO is most suitable for enabling stringent next-gen applications where the target deployment must change dynamically to adapt to changes in the environment, e.g. remote drone control service (in Figure 2) must move alongside the drone, always maintaining the required latency for control.

With the help of its bottom-up, self-organization orchestration design, VineIO enables Piccolo to support fully-decentralized resource allocations in heterogeneous environments. The VineIO wrapper can be embedded within the Piccolo agent, thus making Piccolo services self-organizing, removing any explicit orchestration shim layer (e.g. used by EdgeIO in Section 4.1) from the equation. Furthermore, the information and control message exchange design of VineIO can be leveraged by Piccolo implementations to arrive at a consistent view on the available system resource for enabling robust optimisation.

## 4.3 ICN-based Dataflow System

Information-centric networking (-ICN) is a paradigm that shifts the host-centric Internet into a data-centric approach, with named data objects as the core of the model. ICN enables location-independent communication, in-network caching, etc. We integrate ICN's features with dataflow concepts in the following system to implement a decentralised stream processing system.

ICN-based dataflow system presents a decentralised stream processing framework that builds on top of Named Data Networking [12]. As this approach results in better resource utilization and receiver-

driven communications, it contributes to the decentralisation of the system where scaling decisions, fault tolerance, and orchestration tasks are managed by every actor that belongs to a dataflow.

The following system supports the Piccolo agent implementation per node. This agent will assist actors while taking scalability and recovery decisions by providing the available resources in the infrastructure in a joint optimisation approach. Moreover, we support location-independent naming that spans the resources and communicating actors, i.e., every component in the system is named. There is no need to map to host addresses, whether for communication between actors or during actor placement.

The following sections discuss the general concept of existing centralised stream processing frameworks and the drawbacks, the advantages of using the name-based approach for communications as well as how orchestration and resource allocation are handled in a decentralised fashion.

### 4.3.1 Problem Statement

The Dataflow paradigm is a popular distributed computing abstraction that is leveraged by several popular data processing frameworks such as Apache Flink [13] and Google Dataflow [14]. Fundamentally, Dataflow is based on the concept of asynchronous messaging between computing nodes, where data controls program execution, i.e., computations are triggered by incoming data and associated conditions. This typically leads to very modular system architectures that enables re-use, re-composition, and parallel execution naturally. Most of the popular distributed processing frameworks today are implemented as overlays, i.e., they allow for instantiating computations and for interconnecting them, for example by creating and maintaining communication channels between nodes such as system processes and micro-services with the help of a central orchestrator. These overlay connections do not necessarily follow optimal paths, i.e., the communication flows are incongruent with the logical data flows. This results from the fact that orchestrators may have visibility into compute resource availability but typically have to treat the communication network (e.g., TCP/IP) as a black box.

In some variants of Dataflow, for example, stream processing, it can be attractive if one computation output can be consumed by multiple downstream functions. Connection-based overlays such as Flink typically require duplicating the data for each such connection, incurring significant overheads. Moreover, as a sender-driven system, Flink's inter-task communication requires back pressure or credit-based flow-control system to avoid overrunning downstream consumers, resulting in multiple flow control loops: at TCP and the task manager side, which can be seen as an unnecessary complication.

One key goal for Dataflow systems is to enable parallel execution, i.e., one computation is run in parallel, which also affects the communication relationships with upstream producers and downstream consumers. For example, when parallelizing a computation step, it typically implies that each instance is consuming a partition of the inputs instead of all the inputs. An indirection- and connection-based approach makes it harder to configure (and especially to dynamically re-configure) such dataflow

graphs.

The connection-based approach incurs several architectural problems and inefficiencies as mentioned above, for example, application logic is concerned with receiving and producing data as a result of computation processes but connections imply transport end-point addresses that are typically not congruent. This typically implies a mapping or orchestration system.

Apache Flink, for instance, embraces a central orchestrator referred to as the "JobManager". This JobManager monitors worker nodes, react to failures, handles resource allocation, and maps the dataflow graph to the worker nodes [15]. As mentioned above, the orchestrator's decisions are based on computing resources only. This, combined with the connection-based approach, might result in a less efficient dataflow execution that does not optimally match the logical dataflow graph. In the next section, we present our ICN-based data flow system design that aims to overcome these complexities and avoid the connection-based overlays in the system.

#### 4.3.2 ICN-based Dataflow System Design

The Information-Centric Dataflow system approach supports traditional Dataflow with Information-Centric Networking principles and can be used as a drop-in replacement for existing Dataflow-based frameworks. Its objectives are: (i) reducing complexity in Dataflow systems by removing connection-based overlays and corresponding orchestration requirements; (ii) enabling efficient communication by reducing data duplication; and (iii) enabling additional improvements through more direct communication and caching in the network [16].

It applies a data-oriented approach to Dataflow (not a connection-based one), i.e., computation results are named, immutable data objects and a Dataflow actor is an ICN consumer of its input data and an ICN producer of its output data. Compute workflows are represented in the network as compute graphs used to perform flexible load management and performance optimizations, realized by using Conflict-free Replicated Data Types (CRDTs). Access to computation results happens through regular ICN Interest/Data interactions with the usual properties: strong data authentication, location independence, on-path data caching, and implicit multicast through Interest aggregation and data sharing.

Using Interest/Data allows for a receiver-driven approach that avoids back-pressure and credit-based flow control present in other systems such as *Apache Flink*. The convenience of this approach is that consumers can request data according to its available memory and computing resources as we introduce efficient queue management. Integration of this feature with the receiver driven congestion control results in a stable dataflow that exploits available resources and prevent consumers from resource exhaustion. Moreover, pull-based ICN dataflow model is also feasible for resource constrained devices.

In order to communicate, a Dataflow actor uses application-relevant names in a suitable namespace. There is no mapping to lower layer addresses. Consequently, there are no connections, and the sys-

tem is generally location-independent, i.e., actors are oblivious to the location of other actors, actor locations can change during a distributed program execution, and the system is using the exact same mechanisms to communicate, regardless of whether the system is used in a local network or over the Internet. Location-independence grants more decentralisation to the system as no central orchestrator is needed for mapping between applications' namespace and network addresses. This makes the deployment on bootstrapping, restarting, or scaling simpler.

As we aim for a decentralised orchestration with receiver-driven communications between actors in the system, a synchronization protocol is essential to maintain a consistent view of the compute results between actors of the pipeline. To achieve this, we use the Psync synchronization protocol [17]. In Named Data Networking, Psync protocol is used to address data synchronization in distributed systems. Psync allows producers to publishing their new state in the namespace. In our case, producers publish the names of the newly computed results, and consumers who are already subscribed to the corresponding namespace can get these updates. Then using the published names, consumers create the corresponding interests and send them to retrieve the data back. Management of the namespaces between actors is handled within the compute graph definitions using CRDTs.

One key feature of the system is when the same interest is created and sent by multiple consumers. One data object is needed to serve all the interests by exploiting caching in Named Data Networking (NDN) networks, resulting in a more efficient system with less data duplication over the network.

Each actor uses an output queue to store newly produced data, advertise the data names, and send the data packets to the actors fetching it. On the upstream side, actors also use an input queue to buffer incoming data as a source for the computing function. These queues ensure that the computing functions are always busy computing and never wait for incoming data, thus maximizing the system's resource utilization. It also provides insights for the function from the upstream producers and downstream consumers.

As mentioned previously, with the ICN-based approach, there is no mapping between the application's name space and network addresses of worker nodes which is usually a task handled by a central orchestrator. This mapping adds complexity to the dataflow deployment in different stages. For instance, reacting to a node failure will result in halting the dataflow and re-configuring the actual flow graph across the system. However, in our system, each actor is using the application's namespace for communicating. Thus, an actor failure would not require a change in the configurations as the system is location-independent. This approach allows a fast reaction to the actor's failure or scalability decision. Moreover, by adopting the ICN-based approach along with the Piccolo agent, we can delegate tasks like resource allocation and scaling decisions to the actors and the corresponding Piccolo agent. Every actor can be considered an orchestrator of its downstream consumers, and here it can detect a failure or a slow/fast consumer and trigger a scaling decision without referring to a central node. Also, the Piccolo agent of every node is responsible for the resource allocation requested by actors deployed on this node. With these features, our dataflow system is implemented in a complete decentralized approach overcoming the drawbacks of centralized orchestrations and a connection-based approach.

 

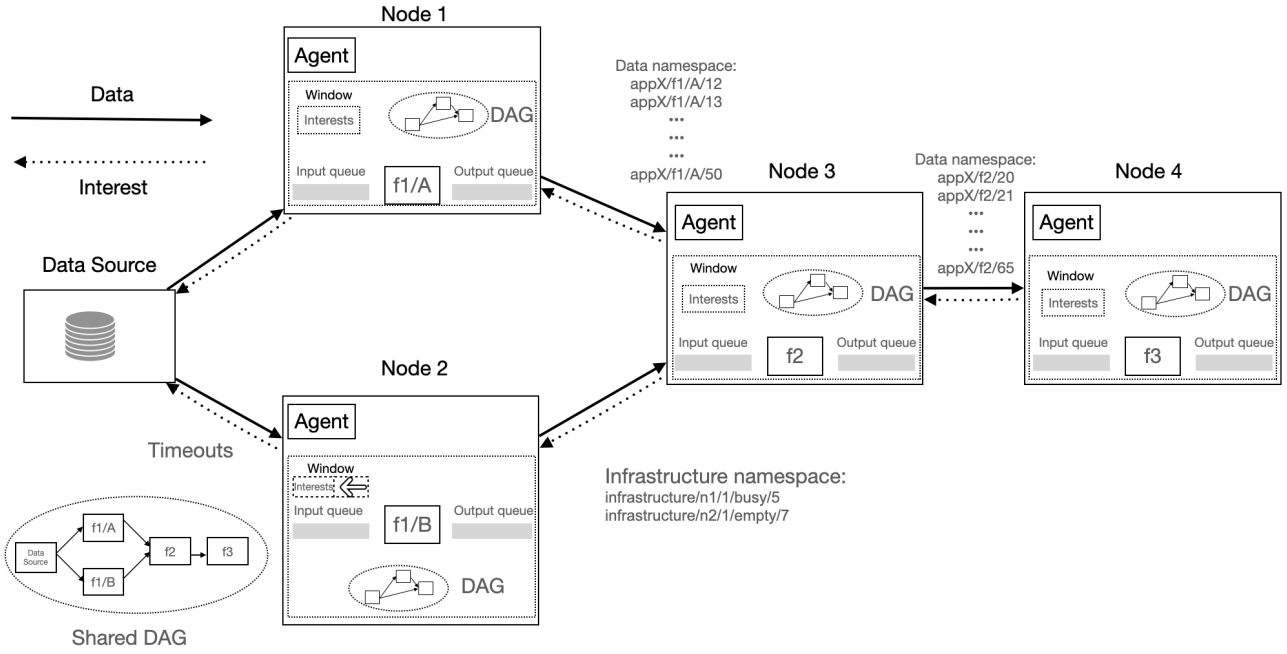


Figure 3: ICN-based Dataflow System

### 4.3.3 Resource Allocation

As shown in Figure 3 every node in the system has a Piccolo agent assigned to it. The agent contributes to the decentralisation of the system as it manages the resource allocation task in the system. This way, instead of having a central orchestrator responsible for this task, every agent in every node is aware of the available resources across the system. It can offer it to the actors deployed on the same node, thus managing resource allocation in a decentralised approach.

A node advertises its resources in a separate namespace using Psync. Thus, every agent holds a view of the infrastructure's available resources. Also, the Piccolo agent gets information from routing protocols. This way, the agent can achieve joint optimization in resource allocation as it decides which available node is the optimal choice for the current scaling mechanism based on the available computing and memory resources, network status, and how far the node is then it offers this resource to the corresponding actor.

Every upstream actor can detect a bottleneck downstream. It can detect a slow/fast consumer, which triggers a scaling decision. In addition, it can detect a failed consumer that requires a new instantiation of this actor. We should note here that deploying a new actor instead of a failed one will not require modification to the flow graph because actors request of named data objects will not change, which will result in less complexity in the system while recovering.

When the actor takes a scaling decision, at this stage, it needs to communicate with its local agent for further insights on the feasibility of the scaling decision, i.e., if there are available resources across the infrastructure and which of these resources are the optimal target for an actor instantiation. When a scaling decision is implemented on a specific node, the local agent is responsible for sending updates

and synchronizing with other agents with its new resource availability.

## 4.4 Hybrid Orchestration for Distributed Computing Architectures based on ICN

In this section, the limitations in centralized and decentralized orchestration of an ICN based in network computing framework is presented. Following this, in order to tackle such challenges in orchestrating and optimizing different aspects of the network, the concept and architecture of deploying a hybrid orchestration model is introduced.

### 4.4.1 Problem Statement

As mentioned in the previous sections (cf. Section 4.1), centralized orchestration frameworks such as Kubernetes have proven efficient in performing cluster management on container-based cloud-native applications. While there are ongoing efforts of porting and enhancing such frameworks to push towards communication infrastructure edge deployments (KubeEdge, AWS Greengrass), the heterogeneous and geographically distributed nature of edge infrastructure, and therefore, the management of service deployments at the edge impose challenges.

Based on a loosely coupled communication model, the location independent addressing of ICNs [18], e.g., using naming schemes to provide access to data directly instead of addressing the host first, is promising to support distributed services at the network edge. Distributed application frameworks extensions on top of ICNs such as Named Function Networking (NFN) [19] and Named FaaS (NFaaS) [20] leverage the location independent addressing of ICNs to provide access to computation results. The subsequent paragraphs uses NFN design rationale to describe the limitations.

NFN uses $\lambda$ expressions to describe a compute workflow as part of an `Interest` packet. Resolution strategies performed during packet forwarding identifies the most suitable node for serving the compute request. Such decisions are taken on a per-request basis at each node using the local forwarding knowledge of the node in a decentralized fashion. Although such a decentralized orchestration is scalable to larger networks, the lack of global network knowledge might result in a sub-optimal decision making, for instance redundant computation of popular functions at neighbouring nodes resulting in an imbalanced resource utilization [21]. Additionally, the name based routing although efficient in decoupling the services from the compute nodes, is highly dependent on the naming structure the consumer uses while requesting computations. As an example, a compute request for execution of a function "func1" over data `"data1"` can be expressed with function first as "/func1(/data1)" or with data in the first position as `"/data1(`$\lambda$`.x call(/func1))"`. This results in a different choice of execution node and hence different completion times and network utilization for different naming structure for the same computation as consumers lack knowledge of the location and size of data and function.

In order to tackle the challenges set by distributed service deployments at the network edge of an ICN, several centralized orchestration solutions have been introduced in the literature such as Amadeo et. al [22]. The authors introduce a centralized orchestration entity in the network following the Software-Defined Networking (SDN) paradigm to exploit the global knowledge of an SDN controller to manage and deploy function at the network edge (e.g., using OpenFlow to manipulate forwarding rules). While the resolution of identifying the most suitable compute node is presented on a per request basis, as in default SDN, we argue that the additional time to consult an SDN controller is sub-optimal for certain Internet of Things (IoT) deployments such as connected vehicle environments with a high degree of mobility.

To overcome this limitation, we present the concept of a hybrid orchestration mechanism for distributed computing architectures based on information-centric networking. The concept employs a logically centralized coordinator that gathers information from compute nodes in the cluster asynchronously to assist the decentralized resolution mechanisms of the compute nodes.

### 4.4.2 Concept of Hybrid Orchestration

In distributed computing architectures based on information-centric networking, such as NFN, consumers request for computation result using workflow expressions as part of an ICN `Interest` packet. The underlying named-based routing and forwarding of ICN takes care to forward the requests towards potential compute nodes. During forwarding, resolution strategies present at each node assist in identifying if a node is suitable to serve the request or to forward it upstream towards other compute nodes in the system. This resolution is performed on a per-request basis directly on the data plane. While this design decision provides a high degree of flexibility in handling requests, the local decision making might result in sub-optimal executions of functions and services as nodes in such systems do not share information about their compute and network load.

The concept of *hybrid orchestration* overcomes this limitation by employing a logically centralized coordinator that gathers information from compute nodes asynchronously, assisting the decentralized resolution mechanisms of the compute nodes. In order to be able to assist the local resolution strategies, the concept assumes the following features to be supported by the network/compute devices within the Piccolo system:

- a handle to access data plane related information such as function request load, functions executed, resource utilization (e.g., from a load balancer, or NFN forwarder).
- a handle to influence compute behavior (e.g., pull a specific function).

Figure 4 illustrates the interactions of the hybrid orchestration mechanism and the compute nodes/-forwarding nodes. During compute request forwarding, an orchestration entity monitors compute and forwarding nodes in the entire network under its scope over a period of time. Obtaining a global view of the network, the orchestrator is able assist resolution strategies, for example, by instructing compute/forwarding nodes to pre-fetch function byte code or data, instantiate or terminate function

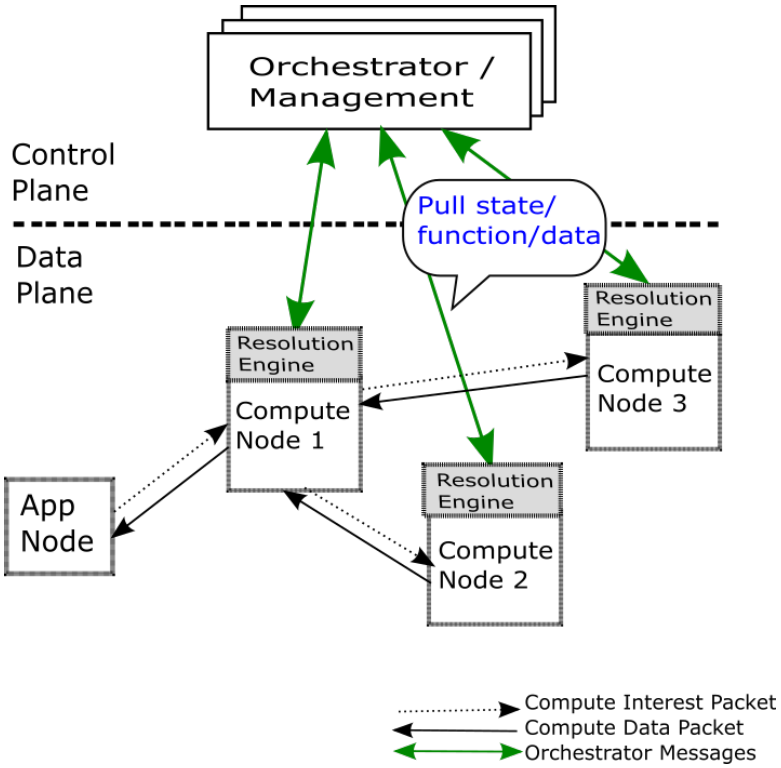


Figure 4: Hybrid Orchestration Architecture for ICN based Compute frameworks

executions. Forwarding tables are modified by the orchestrator as well to ensure that ICN Interest packets are forwarded towards the right compute nodes.

In difference to the centralized orchestration solutions where the forwarding nodes communicate with the orchestrator for every compute request resolution, the hybrid orchestrator operates asynchronously to the existing resolution strategies by polling the network devices periodically on the control plane and providing optimization suggestions to nodes.

Therefore, a control protocol, which is aligned to some of the Piccolo APIs, for hybrid orchestration is developed operating on the following three phases.

- **Compute Node Discovery:** Every time a new compute node joins the network, it registers to the hybrid orchestrator using a specific name prefix representing a specific namespace (e.g., `"/orchestration/register/NODE_ID"`). During registration, the compute nodes provide a unique node identifier NODE_ID which allows the orchestrator to directly access data of a particular compute node. This design is similar in principle to the Named Link State Routing protocol of the ICN implementation - Named Data Networking [12], used to identify NDN forwarders in the network [23].

- **Monitoring Compute Nodes:** The hybrid orchestrator, periodically monitors and polls the compute nodes using their unique identifier to gain knowledge of the function demand, execution distribution, resource utilization at each compute node in the network. Based on all the information gathered from the nodes, the orchestrator can propose changes or actions to the

nodes (e.g., `/orchestrator/NODE_ID/OPERATION` while an operation might be fetching the *status*.

- **Enforcing optimization suggestions:** Based in the optimization goal, the orchestrator can suggest some actions to the nodes if there is a scope for improvement. For instance, if the completion time of a computation can be reduced by instantiating it closer to the consumer on less loaded compute nodes, the orchestrator can instruct the compute node *load* and `instantiate` the function. Further examples of such optimization decisions could be enabling or disabling a function at a node, or fetching the popular data in off-path caches.

#### 4.4.3 Preliminary Evaluations of Hybrid Orchestration

In order to verify the benefit that hybrid orchestration brings to an ICN based in-network compute framework, we evaluate using simulations (cf. Section 6.1) the effect on reduction in completion times with the presence of the hybrid orchestrator against an NFN network. Completion time is defined as the time taken for resolving the consumer's compute interest and responding with the result of execution in a `data` packet. The simulation results show a reduction in the average completion time with the presence of an hybrid orchestrator. This reduction in completion time can be attributed to the decisions taken by the hybrid orchestrator to pre-fetch functions and instantiate them at compute nodes closer but off-path to the direction of the packet upstream. Hence the distance travelled by the interest to reach a suitable node for execution is reduced.

Additionally, the compute nodes with the available function and data may not execute the functions right away due to the resources being blocked by other parallel computations. Such interests have to wait for the resources to be freed and executed increasing the completion time. The hybrid orchestrator identifies such heavily occupied nodes and pre-fetches functions at its neighbourhood to handle dynamic compute demands. This reduces the waiting time of the interests as well as balances the execution load on the compute nodes.

#### 4.4.4 Discussion

The introduction of an orchestration entity in the network introduces additional communication overhead for the control traffic in comparison to purely decentralized orchestration as presented as one of the Piccolo research challenges (see Piccolo Deliverable D3.1 [3]). With an increase in the deployment of compute nodes, the control traffic is expected to increase as well. Furthermore, orchestration entities monitoring the network devices will also increase, resulting in additional overhead. However, the shorter the monitoring interval is, the more accurate are optimization decisions from the orchestrator. Depending on the needs of the application, the algorithmic model at the orchestrator can be implemented targeting the optimization of certain aspects of the network such as completion time of compute requests, resource utilization, bandwidth efficiency and energy consumption.

 

In summary, distributed computing architectures based on information-centric networking use decentralized resolution strategies to identify the most suitable compute node for a compute request. While such a solution provides a high degree of flexibility to handle compute requests, the limited local knowledge of resolution strategies to execute or forward a request might lead to sub-optimal decisions, e.g., an increase in the completion time of a request. We presented the concept of hybrid orchestration for distributed computing architectures based on information-centric networking that employs a logically centralized coordinator that gathers information from compute nodes in the cluster asynchronously and assists the decentralized resolution mechanisms of the compute nodes. The design of the solution addresses aspects of Piccolo's joint optimization research goal by taking compute loads of node as well as the topological-aware deployment of applications and functions into account, and supporting location-independent servicing at the network edge. We implemented the concept in simulations and compared it against the Named Function Networking architecture. Simulation results have shown improvements in the completion time of computations between 16% and 51% for an increasing number of compute nodes in an hierarchical network deployment compared to the default performance of NFN.

## 4.5 Orchestration for Smart Factory

The Smart Factory use case requires an optimized placement of IEC 61499 function blocks (Piccolo functions) given by means of a data flow graph within a network topology (see figure 5). This placement is primarily based on the location of actuators and sensors for certain functions but also on compute capabilities and topology considerations like node neighbourhood. While such a placement happens first when a program is injected into a node of the network topology it can also happen dynamically on various topology changes like network link changes or location changes of sensors. Hence communication between different components of the overall distributed system should be location-independent to remove addressing complexity. In the light of these challenges the Piccolo infrastructure in the Smart Factory use case can be described as a culmination of ideas partially described in 4.2 and 4.3. There are, however, many simplifications and also differences.

### 4.5.1 Addressing Aspects

Since Erlang/OTP [24] is used as technology foundation there is already a lot of utility present to manage communication between agents and functions since both are modelled internally as Erlang actors. In particular Open Telecom Platform (OTP) provides a TCP based protocol for native communication between Erlang entities and a distributed process registry which enables the communication between named processes based on group membership and so called scopes. Such scopes can be used to define overlay networks which make them especially interesting when multiple layers of processes are involved which serve completely different purposes (e.g. function placement vs. work piece management). The use case currently has no need for hierarchical namespaces for communication between Piccolo agents of functions, even though that can become a requirement later on. The process registry

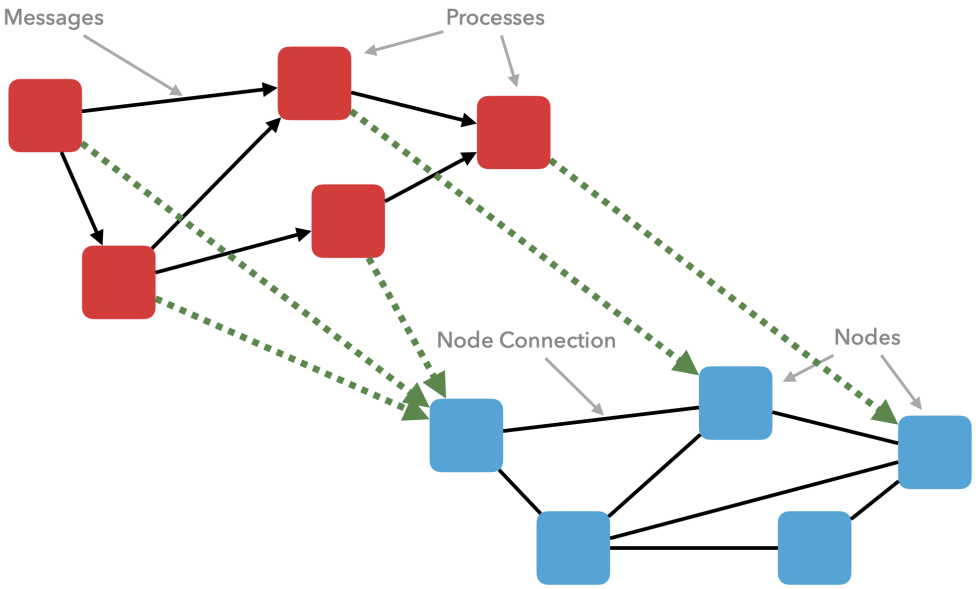


Figure 5: Mapping of IEC 61499 function blocks to network nodes

implements strong eventual consistency.

IEC 61499 applications are structured into composed blocks which represent groups of function blocks (see figure 6) which again are modelled as an actor within Erlang. Hence the nodes of a IEC 61499 data flow graph can be mapped quite naturally to this process registry. Note that there is no central management entity involved, the process registry is completely distributed. Compared to the ICN design 4.3.2 this overall approach is still connection-focused (even though location-independent) and sender-driven but simplified ICN like semantics can actually be achieved by using membership in process groups for interest in named data.

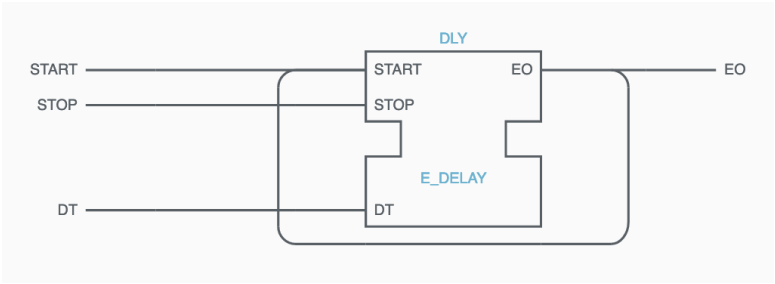


Figure 6: Visualization of a IEC 61499 function block

### 4.5.2 Optimization Aspects

As function placement is a crucial aspect of the Smart Factory use case it is only natural to be aware of the topology state, node resources and possible changes. Initially existing routing protocols can be used to accumulate topology information (IS-IS), but it is also possible to move routing functionality completely onto the application level within Erlang. It should be stressed here that link-state protocols

are necessary to be used since they offer a view on the whole network topology on every node. For distributing node resource information and the current placement of functions the Erlang native distributed database Mnesia [25] is used initially. Both sources of information are part of an internal information base which allows every Piccolo agent maintain a digital twin of the factory line which is also used for work piece management. Further optimization layers can observe the information base and the digital twin to execute changes in function placement. Since every agent is aware of changes in the information base and is also capable of executing changes in function placement, it is necessary to elect a leading agent (see figure 7). Note that optimization results are independent of the node since the information base contains the same data everywhere due to the usage of distributed transactions in Mnesia. This prevents in particular optimization cycles which may happen when different nodes have different world views. However, Mnesia is not suited in the long term for critical applications since it's not partition tolerant and also not capable of handling complex topologies. It is probably desirable to unify the distribution of all information into a single protocol later on.

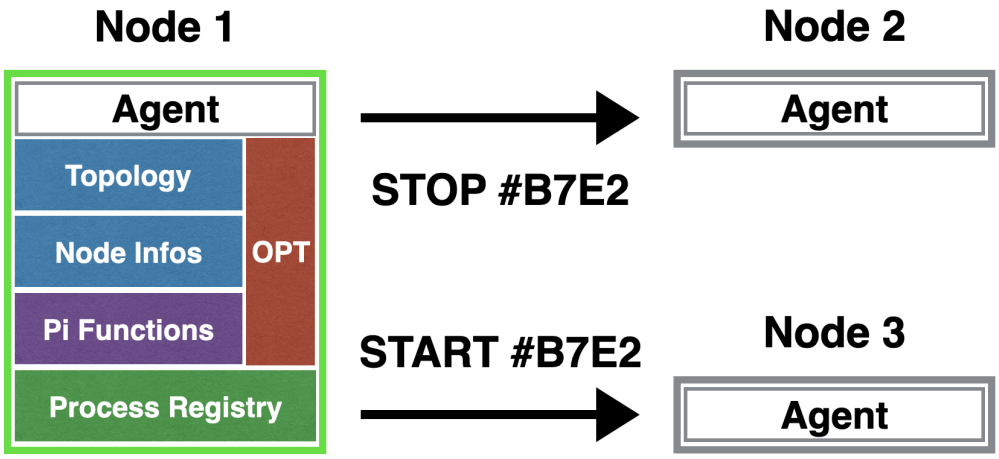


Figure 7: Elected leader agent performing function migration

Optimization efforts are based on SLAs which are attached to each function but also on topological properties of the IEC 61499 data flow graph such that for example neighbourhood between functions in the data flow graph can be preserved. Hence there is also a SLA for the data flow graph itself. The function specific SLAs contain for example connectivity requirements to sensors or actors while the data flow graph SLA contains rather abstract optimization objectives like the mentioned neighbourhood properties. The agent is supposed to enforce these SLAs based on efforts of the optimization layers. In a real world factory there will actually be two different types of functions when it comes to optimization: rather statically assigned functions for actuators and sensors, and rather dynamically placed functions which need no access to hardware. The reason for that is obviously the often static placement of related hardware.

In the broader picture the function placement optimization is part of a larger system which also deals with aspects of the work piece traffic management. The latter is not further discussed here since it is very use case specific.

 

#### 4.5.3 Byte Code Distribution

Every node has all the Erlang byte code of the IEC 61499 application locally stored—even the byte code for functions that the node itself currently does not execute. This also allows every node to reconstruct the data flow graph of the IEC 61499 application for optimization. The byte code distribution can be achieved through the Erlang code server which allows to share byte code initially when the IEC 61499 application is injected from one node using Remote Procedure Calls (RPCs), updates are also possible later on. Hence migrating byte code of functions is not an issue.

## 4.6 Summary

The interpretation of the term edge computing today is to a great extent limited to bringing executions and computations closer to the consumer. Different available solutions used to this end today e.g. KubeEdge or ioFog are ineffective due to their strong assumptions on the underlying communication infrastructure. In this section, we have presented work that tackles limitations of today's communication and compute infrastructure deployments such as scalability challenges of orchestration solutions (cf. Section 4.2 and Section 4.4), location-independent addressing, and flexible data flow processing (cf. Section 4.3 and Section 4.5) and their potentials to optimize the utilization of compute and network resources within a Piccolo infrastructure.

# 5 Resource management

Resource management is a key issue to solve for in-network computing. Resources (whether for computing or communications) need to be allocated to the different tasks, nodes and tenants, in a way that is reasonably optimal and resilient. The key concepts, state of the art and architectural approaches are summarised in Del3.1 [3]. The following section presents some solutions that we are experimenting with. They are in the context of different environments and orchestrator approaches presented in the previous chapter, but should have wider applicability. Our first approach uses decentralised, autoscaling, within an ICN-based dataflow system. Our next approach uses an auction-based marketplace, within the Trusted In-Network Computing (TINC) platform that is being used for the 'risk management' proof of concept demonstrator. Our time-sensitive networking work also studies the problem. Future work will progress these experiments and seek insights that can lead to a more general model and possibly to corresponding API updates.

## 5.1 Decentralised autoscaling within ICN-based Dataflow System

This section explains the decentralised scaling approach adopted in the system. After describing the role of the Piccolo agent in 4.3.3 in the system, here we give more details about the autoscaling mechanism, the tasks done by the actors, and how the agent is involved in the procedure. It provides more information about how scaling decisions are triggered in a decentralised paradigm and what parameters are taken into account. In this model consumers retrieve data from the producers by exchanging interests, which can be regulated by the congestion control. In the following sections, we describe both the scaling decision making and congestion control mechanisms.

### 5.1.1 Autoscaling

Scalability is a task handled by the actor and the agent. Every actor is responsible for the scalability actions of its downstream consumers. As mentioned in 4.3.2, every actor maintains input and output queues. The output queue is not only about storing computed results of the actor. This queue can provide significant insights to the actor from its downstream consumers. These insights could detect slow/fast consumers or even detect the failed ones.

Depending on the resources and output type, a configured threshold is set for this output queue. When this threshold is reached, a function cannot process more data until performing some actions. A full queue is interpreted in two ways by the actor. Either a consumer has failed, and this can be further recognized from the loss of ACKs sent by the consumer to its upstream producer, or a consumer is slow, and scaling should be triggered. At this stage, the actor sends a scaling request to the local agent that can choose the optimal resource (if available) to offer to the actor. Agents across the nodes use Psync to synchronise the available compute resources in a different namespace than the compute results. This way, every agent is aware of the compute resources across all nodes. We refer

 

to the compute resources as compute slots the agent uses to decide where to instantiate the new actor. Furthermore, agents also take network resources information into consideration for decision making, i.e., link capacity, available bandwidth, and forwarder resources.

Then the local agent will trigger the agent of the target node for the actor instantiation. An actor instantiation on a node requires: the application's sync prefix to subscribe to, the routing graph and the current compute graph, its prefix, and its functionality. When triggering a scaling decision, the actor should also produce watermarks to its downstream consumers. When receiving these watermarks the downstream pipeline's computation should stop for some time until receiving a new signal watermark to resume the computation.

### 5.1.2 Receiver-driven Congestion Control

To avoid overloading the resources in our system, an additive-increase/multiplicative-decrease (AIMD) congestion control mechanism is implemented by each actor. Actors hold a dynamic window of pending interests. This window would increase/decrease based on the timeouts and data packets it receives back. However, the window size cannot exceed the available space in the corresponding actor's input queue. The input queue, which buffers the input data received and not processed yet, reflects the actor memory and compute resources and how fast data is consumed from this queue. Thus, when the pending interest window has the input queue as a maximum limit, it does not send interests based on what it receives back only, but also it depends on the local resources. This results in a more stable pipeline and avoids overloading the actor's resources. When this window becomes far behind the upstream updates, the upstream producer can detect this and trigger and instantiate a new instance of this slow-consuming actor.

## 5.2 Auction-based marketplace within the TINC platform

TINC is a Trusted In-Network Computing platform that combines concepts of serverless computing, Trusted Execution Environments (TEEs) and overlay protocols in order to provide shielded and outsourced computations. While in Piccolo Deliverable D2.2 the characteristics and architectural elements of a TINC node were described, the following sections present the overall architecture of the TINC platform and highlight the resource allocation feature that is done in an auction-based marketplace and could potentially be applied to the Piccolo architecture.

In the context of TINC, nodes have two different roles:

- **Regulars**: the simplest form of a TINC node that is capable only for (secure) computation tasks and may not always be available - referred as TINC nodes. These nodes are connected directly or indirectly to Mediator nodes in order to receive function tasks.
- **Mediators**: similar to regular TINC nodes but with the responsibility of being always available

in order to offer relay properties to regular nodes that cannot be directly connected and also form markets (marketplaces) for resource allocation as presented in the following sections. Note that each mediator can have only one market.

The system assumes that there is an infrastructure owner that is called In-network Computing Provider (INCP) [5] and offers computation elements. Additionally, there are Application/Service Provider (AppSP) that generate/develop applications/functions and also data owners that have/gain access to the post-processed data (data owners can also be application providers). To trigger computations on a TINC node the AppSPs use a service called TINCmate in order to produce manifest files (see Piccolo Deliverable D2.2) that describe functions and can be consumed by TINC nodes and start the function execution. The function is considered as a secure Docker container that executes the function task inside an Intel SGX enclave [26] to protect it from unauthorized access and manipulation.

The major element of a node (both mediator and regular) is the TINC-engine (or TINC) that is a daemon process, which connects the current node with other nearby nodes and is responsible for implementing various TINC operations like resource allocation, computation configurations, outsourcing of incoming tasks and bootstrap function processing. Each Mediator forms its own marketplace which also advertises. Regular nodes can subscribe to a market in order to be able to receive computation tasks. The protocol used by all the TINC nodes is "GossipSub" which is an extensible pubsub protocol, based on randomized topic meshes and gossip [27]. The aforementioned stakeholders and the overall architecture is depicted in Figure 8.

Each Mediator node can optionally include an attestation service that is responsible for ensuring the correctness and freshness of the application code and data while also creates sessions for each computation in order to maintain and transmit each function's secrets in a secure fashion (see section 5.2 Deliverable 2.2). Note that this service is always available as a component to TINCmate and a TINC node can communicate with that attestation service in order to receive functions' secrets and evaluate the measurement of a function to be executed in a platform. Therefore it is crucial to point in the manifest file the agreed attestation service.

The TINC-engine consists of two major components which are used for the inter-communication of the nodes and the actual orchestration and function placement procedures. The first component is the *overlay agent* which covers operations like node discovery, identification and resource advertisement. The *overlay agent* is also provided as part of the TINCmate service since it presents a topology to the service in order to choose a node to send function deployment requests. The second component is called *machine* and is responsible for the resource allocation, function configuration and task outsourcing.

[5] An INCP is similar to an Infrastructure Provider but with specific responsibility in offering computation instead of networking operations

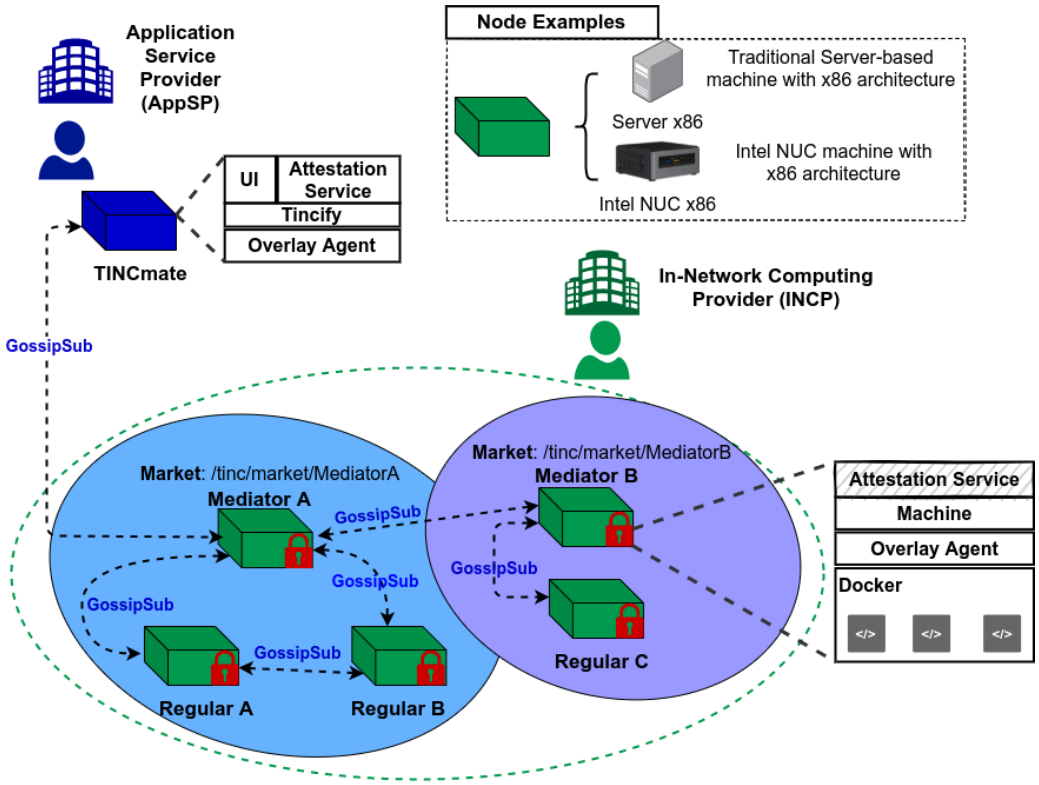


Figure 8: TINC system architecture

### 5.2.1 Resource Allocation through Auctions

In Piccolo Deliverable 3.1 we described a novel market-based resource allocation mechanism [28] that leverages from Vickrey-English-Dutch (VED) auctions [29] on the micro-level in order to provision resources over a geo-distributed set of nodes. Each node can potentially act as a market holder in which the available resources (in the case of TINC these are called functions/Execution Environments (EEs)) are offered as items while the users that want to execute/acquire those resources are called bidders. The auction's purpose is to derive a price vector along with item-bidder assignments that indicate a competitive equilibrium, i.e. the auction has to satisfy the bidder's demand (and requirements).

The ultimate goal of the operation is to allocate the available resources based on demanded and supplied values (bids and prices) in order to reach an equilibrium in the market (i.e. the bidders cannot acquire their assigned items for a lower price in any other competitive equilibrium). The repetitive nature of this mechanism (i.e. the market needs to always reach a stable state) allow to reduce the execution time of the procedure. For instance, if the supply and demand conditions of the market remains the same, the equilibrium prices of the next time-slot will be similar to the previous ones and therefore the auction will terminate immediately since it initiates from the equilibrium prices. Moreover, the mechanism assumes truthful bidders since they collect an item in its minimum possible price and do not deploy complicated strategies that would lengthen the execution of an auction.

In TINC nodes can have two modes of execution in terms of resource allocation: **participant** or **auctioneer**. In general, the participant is a node that subscribes/participates to a market and provides its resources, while the auctioneer acts as a higher rank node that receives clients' requirements and executes auctions. In more detail, a market is owned by an auctioneer or a group of auctioneers[6] that belong to one INCP.

Markets are topics (e.g. from publish/subscribe mechanisms) where participants would advertise their resource map. A resource map is a combination of static and dynamic resources and even properties of a node that makes them available in a market that participates. Examples of such resources are the ability to handle sensitive applications (i.e. execute tasks in a TEE), CPU and memory information, available functions/EEs and also currently active functions/EEs. Those details are not only mapped to aid the auctioning procedures but also sorted based on a significance level that is derived from a node owner (INCP). Pricing of such resources are left to the node owners and are given upon bootstrapping a node[7].

Based on this resource map each node shares a graph of resources and awaits a client's request for allocation. The procedure of accepting a request is divided in three phases:

**1. Resource Discovery**: Discovers which nodes meet the application's requirements and ultimately would introduce balance on the market. An important step prior to receiving client requirements is to bootstrap the network, i.e. identify the markets to connect to in order to exchange resource maps. This step is achieved through a public marketplace which acts as a list of available markets. That list is used by the auctioneers to publish their market details. On the other hand, the participants will identify the markets that suit them better and connect to them (i.e. subscribe). For a participant to pick an appropriate market is primarily a configuration step that is performed by the node owner(INCP). After the subscription of a participant to a market[8] it will receive resources of other participants in order to form its resource map.

**2. Allocation**: The allocation step involves the creation of a candidate list among the participants of the market that an item (function request) is auctioned. The candidate list is a sorted set of nodes that meet the requirements of a function. This list is formed after a function request has reached the auctioneer and an auction has been initiated. Note that the function request is the TINC manifest that includes also a bid value for the function requested. The winner is the first candidate on the list since this node will not only meet the requirements of the application but also contribute to achieving an equilibrium in the current market. The function request ultimately is handed over to the "winner" node.

**3. Configuration**: The configuration step involves the actual invocation of the function/EE[9] after it has been received by the "winner" node. This step is considerably influenced from the virtualization

[6]The auctioneer nodes are set by the INCP but by default each mediator is an auctioneer.

[7]Resource pricing: TINC uses the function/EEs as a resource price, i.e. each function/EE has a pre-configured initial price that is adjusted each time an auction lifecycle ends.

[8]participants of a market can be subscribed to other markets

[9]The EE is a concept of Piccolo that is inherited by TINC to serve Docker Images as execution environments. For more details see Deliverable 2.1

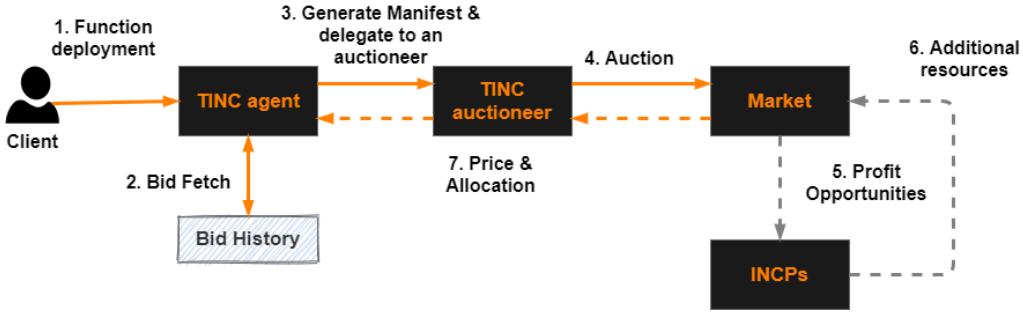


Figure 9: TINC resource allocation mechanism

technology used and the boot configuration time it requires. In TINC we use Docker containers that impose a mediocre execution latency (depending on the function). However, since TINC utilizes TEEs and provide attestation features the execution time is also affected by the attestation protocol and the limitations of the TEE technology that is used[10].

In Figure 9, we depict the provisioning procedure of TINC upon the arrival of a new function deployment request in the market. At first a client sends a request to its local market. The request is directed to a corresponding TINC agent (step a) that is a small service that generates manifest files based on function requirements and bids on behalf of the AppSP. This agent will act truthfully and set the bids equal to the actual gain, i.e. meet the function requirements. To set those bids a history of bids is taken into account (step 2) and ultimately generate the manifest file and delegate it to an auctioneer (step 3). The next step (step 4) involves the execution of a VED auction and a feedback phase about profit opportunities to INCP (step 5) in order to engage additional resources to the market (step 6). Steps 5 and 6 are optional and are not executed for every auction. Instead, the indication of opportunities is done in intervals. The final step (step 7) concludes with the allocation of VM resources for the client at a specific price. The last step is shown in dashed line as the auctioneer forwards this information to the agent that initiated the actual function deployment. This is done in order to periodically inform the AppSP about its expenses in the market.

## 5.3 Hardware resource management

Hardware resources can be grouped into different categories:

**1. Memory**: this resource is easy to manage. Hardware memory is limited for data storage, code memory and also entries in tables such as hash tables, Content Addressable Memory (CAM) entries lookup tables etc.

**2. Compute Resource**: this resource is well known with measures such as Mega Instructions Per Second (MIPS) or Floating point operations per second (FLOPS) and others. However, in practice it is difficult to assign a MIPS value to a process and to estimate the remaining processing power if 5 tasks are already running.

[10]The TEE technology that is used is Intel SGX and has an application physical memory(protected) limitation of 93MB ([30])

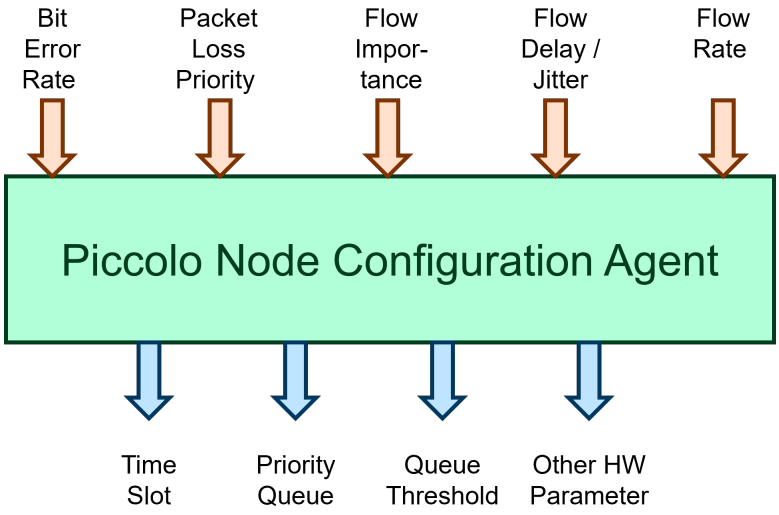


Figure 10: Piccolo Node Configuration Agent translates Flow Requirements into HW Parameters

**3. Capacity**: this limited resource is related to the transport capacity of a link, a transmission line, an on-chip bus etc. It is a data rate measured e.g. in bits per second or packets per second. In case of time-slotted transmission such as T1/E1, Synchronous Digital Hierarchy (SDH) the capacity is translated to a given number of slots. The real-time Ethernet technology Time-Sensitive Networking (TSN) is a mixture of both, transport capacity and time-slots.

While items 1 and 2 are well known, management of item 3, the capacity, is in focus within this project. Both depend on each other. Reserved time slots take bandwidth from the packet based transmission and increase the packet jitter of the statistical traffic. The task of hardware resource management is done by a Piccolo Agent as shown in Figure 10.

The task of the Piccolo configuration agent is not trivial, as a new flow shall not influence the parameters of existing flows with already granted QoS parameters. This must be checked for every new flow by an acceptance algorithm.

To solve this challenge a practical approach will be followed:

1. a default configuration of time slot period and statistical period will be worked out.

2. a reasonable number of slots for the time slot period will be specified and suitable bandwidth granularity, e.g. 100 kb/s as ‘bandwidth token‘ for the statistical period.

3. a small number (e.g. 4) of QoS classes will be defined and default number of bandwidth tokens assigned to each.

The goal of this procedure is to replace complex bandwidth-versus-delay calculations by simple assignment of bandwidth tokens. Acceptance of a new QoS flow will then be simplified to checking if there are enough free bandwidth ’tokens’ in the respective QoS class. For the TSN class, the check will be for free time slots.

# 6 Other Elements

In this chapter we cover some other miscellaneous topics. Firstly, a description of the simulation tool developed to evaluate the hybrid simulator in Section 4.4.3. Secondly, a summary of the 'telco edge cloud' concept, which we have contributed to at the GSM Association (GSMA) - the idea is a federation of operators that expose their edge compute resources in a common, consistent way; the concept can be extended to include in-network computing. Finally, some experiments to ascertain whether an operator's CV-OAM capability could be provided using Erlang.

## 6.1 In-Network Compute Simulator

In this section, we provide a description of the development of a simulation tool available to evaluate orchestration and management mechanisms with respect to in-network computations/function deployments based in Information-Centric Networking. The tool is used to evaluate different aspects of in-network computing systems and was used to generate the results presented in Section 4.4.3.

Within the ICN networking community, there have been several simulation tools introduced in the past including ndnSIM [31] to simulate the behaviour of a Named Data Networking deployment. ndnSIM is an extension to the famous network simulator tool ns-3 [32] that supports an NDN specific protocol stack bundle. It allows to combine well established network functionalities (e.g., wireless communication, Internet Protocol support, etc.) supported by ns-3 together with aspects from Named Data Networking. Based on the presented simulation tools, we developed incSIM [11] – in-network compute simulator – as an extension for ns-3 making use of principles from ndnSIM as well. As of now, incSIM allows to define stateless, monolithic functions/computations deployed within a tuneable network topology, and to evaluate effects of those deployments in the overall system. Examples for such effects include the completion times of functions/computations regarding their deployment as well as investigations on the effects between host-centric (e.g., Internet Protocol) and data-oriented (e.g., using NDN protocol as part of ndnSIM) deployments. As in today's version of incSIM, the simulator supports features to explore the orchestration and management mechanisms and analyse their effects on the deployment structure. Examples of parameters to be modeled within the incSIM tool include:

- option to define different entities: data, functions, data/result consumer, compute nodes, data producer (data to operate on), and function producer (byte code)
- option to influence the behaviour of entities: request load patterns, compute capabilities of individual nodes, function properties required for execution, network link properties, etc.
- option to define different topology deployments of the mentioned entities provided by a topology generator (today limited to hierarchical deployments with point-to-point connections)

[11] https://github.com/boschresearch/incSIM

- option to define initial deployments of functions on compute nodes

As the incSIM is based on ns-3, it uses the abstraction model of that network simulator. All nodes within the ndnSIM are based on the ns-3 model of a Node object. This object was extended to INC Node for simulating incSIM compute. In incSIM, nodes can be configured as consumer nodes requesting for data or a compute result, compute nodes hosting and executing functions, function providers providing access to byte code representations of functions/computations, as well as data providers providing the actual data to apply to a function/computation. In the current simulation set up, the compute consumer nodes are also configured as data providers simulating the behaviour of user edge devices. Figure 11 illustrates three examples of the implementation of nodes within incSIM.

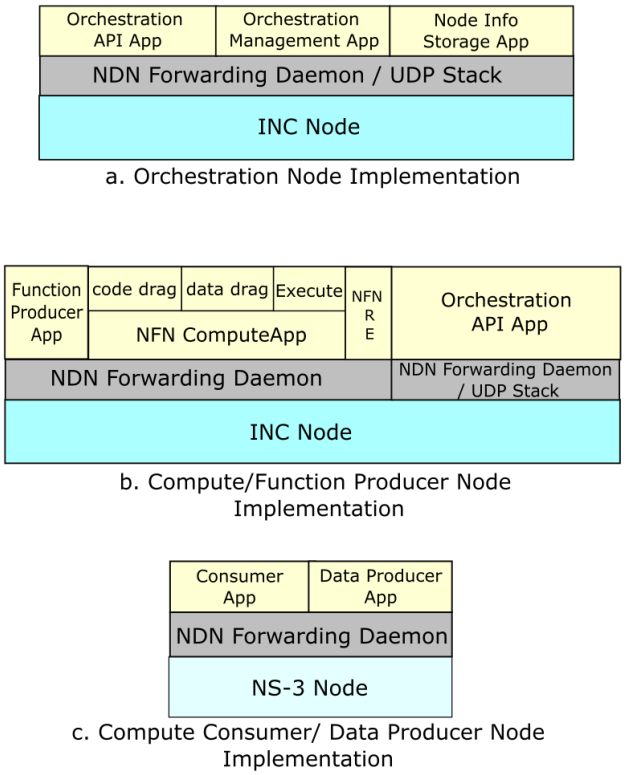


Figure 11: The different node types and their configuration in incSIM

The data/result consumer nodes (see Figure 11 (c)) raise (compute) requests according to a request configuration pattern tuneable within the simulator. Requests are processed by network forwarders and compute nodes until the most suitable node for execution is identified. In the literature, there are already proposals for distributed computing frameworks on top of ICN, including Named Function Networking [33], Named function as a service (NFaaS) [20] or Compute First Networks [34]. To achieve this, we implemented the resolution mechanism of NFN (further details are in [33]). The NFN resolution engine is present on each compute node responsible to identify the most suitable node for the processed compute request based on the availability of the local data and function byte code. The supported resolution engine is implemented in a modular fashion which allows to replace

the mechanism with other proposals in the future, e.g., NFaaS or in-network compute mechanisms. If compute nodes require additional data/information such as the byte code of the function to be executed or the data for which the function should be applied to, the compute nodes are able to fetch those data from function/data producers in the network.

Besides the support of simulating the effects of function/service deployments, the incSIM focuses on mechanisms to improve the overall resource utilization in the overall system including network and compute resources. Therefore, an orchestration module (Figure11 (a)) is available to investigate the influence of orchestration algorithms (here: centralized orchestration mechanisms are supported right now). The module consists of two types of applications: (i) a centralized orchestration entity that executes optimization algorithms, and (ii) an orchestration API entity present at each compute node to be able to share compute node related information (e.g., functions hosted, packets processed, etc.). The monitored information is made public available in the system (e.g., for a centralized component) by offering two control plane interfaces accessible via both the NDN transport protocol or the UDP/IP transport protocol. In this version of the orchestration module, the centralized component frequently fetches status information from all compute nodes in the system which can be configured in the simulator as well. Based on this information, the orchestration/management application can execute specific orchestration algorithms targeting different optimization goals. Current implementations offer a modest orchestration strategy to reduce completion times of computations in the system by bringing them closer to the consumer and enforces them using the control plane API offered by the orchestration API.

In order to investigate the performance and overhead of every application in incSIM, different trace components are available. Examples for traceable data includes the consumer response ratio, compute request-response round trip time, computation load at compute nodes, or the orchestration packet overhead for status information sharing as well as action enforcement.

## 6.2 Telco Edge Cloud - GSMA's Operator Platform

The concept is that all mobile operators should expose their edge compute resources in a standardised and consistent way, so that enterprises and application developers can more easily access edge cloud offerings. Thereby, operators can better serve the global demand for innovative, distributed and low latency services.

From the Piccolo perspective, our belief is that this concept will be (and should be) extended to in-network computing. The telco edge cloud can be seen as an initial step towards in-network computing. It is also a test case about how a federation of operators can work - whether it's simply about a common technical interface, or whether it also includes a deeper technical or commercial relationship.

The concept has been developed by GSMA, which is an industry organisation that represents the interests of mobile network operators worldwide. GSMA calls the commercial concept the "Telco Edge Cloud" and the technical capability to achieve it is called the "Operator Platform".

 

The Telco Edge Cloud is particularly targeted at enterprises. GSMA states the particular benefits that the operators can bring (implicitly, compared to the hyperscalers): the vast local footprint of operators, meaning that a service can be provided locally, so as to improve its quality and reduce the amount of data uploaded; the operators' excellent position to deliver on data sovereignty principles and to preserve the endusers' privacy; their competence to provide high-reliability services; and so on.

The Operator Platform's architecture and requirements are introduced in a GSMA whitepaper, whilst a "permanent reference document" describes the next level of detail.

The most important conclusion is that there needs to be a simple and universal way for Application Providers to interact towards the edge computing platforms. This NorthBound Interface (NBI) exposes the edge computing capability, integrated with the network services (meaning the capabilities that a mobile operator already provides, such as security, mobility and managed connectivity, and knowledge about the enduser, such as their location). A universal NBI ensures that application developers can "write once, deploy anywhere", onto any operator.

There also needs to a common interface between operators East West Bound Interface (EWBI). This allows operators to federate (cooperate) - so that a customer (of one operator) who has roamed will access the edge compute of the nearby operator ("local breakout").

Figure 12 shows the main interfaces that GSMA has defined. The GSMA has also defined the architectural approach, for example a "separation of concerns" of the OP and application providers - which means that the latter requests the service, with some quality of service parameters, whilst the OP decides the best way to deliver it (for example, which edge nodes are used).

The GSMA also proposes that the best body to do the actual standardisation of the APIs and the functionality to achieve the various requirements is 3GPP, with cooperation from ETSI MEC for some aspects of the edge node. This activity is underway. Further, there have already been some trials, whilst more are under discussion.

Piccolo Partner BT has been heavily involved in the GSMA's efforts. We have also described the NBI using Swagger, which is a tool that implements the OpenAPI spec, and is becoming popular as a way of describing, producing, consuming, and visualizing REST interfaces.

Further details about the GSMA's work - whitepapers, documents and videos about trials - can be found via their webpage [35] .

## 6.3 Erlang experiments - Erlang for CV-OAM

In this section, we provide a description of the experiments conducted to investigate whether Erlang is suitable for implementing a CV-OAM environment that will serve or be provided by a Network Provider. Erlang was evaluated by investigating its built-in capabilities that could support CV-OAM

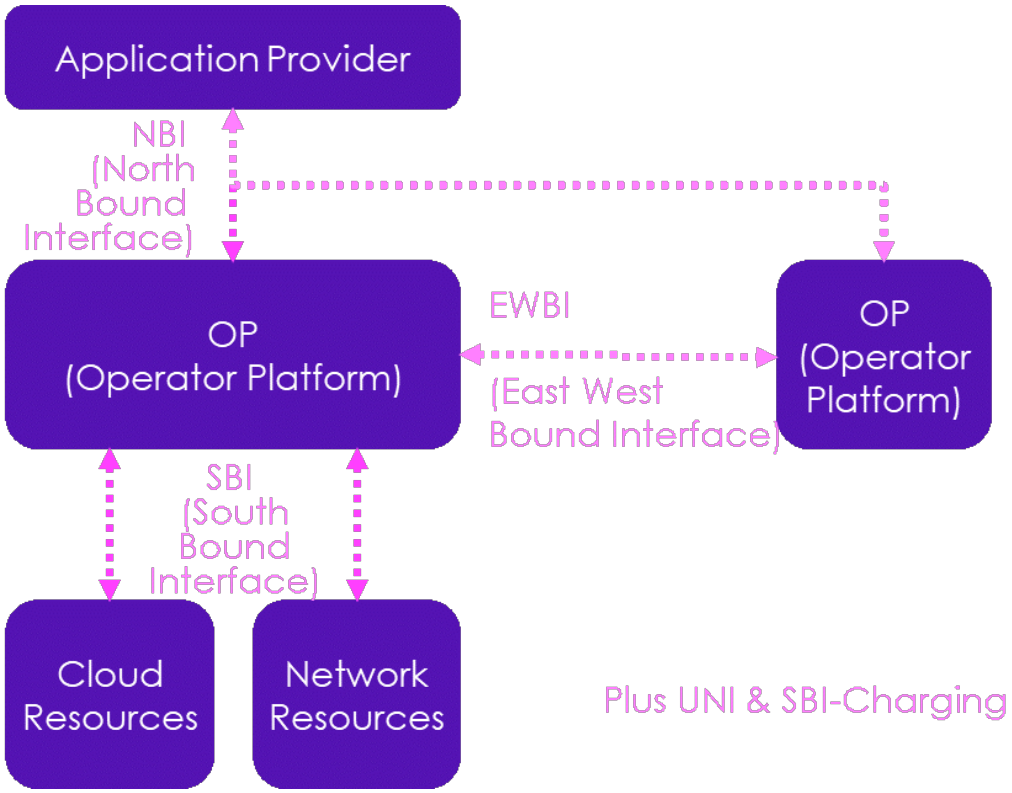


Figure 12: Operator Platform high-level architecture for edge computing

and by using it to create code that supports CV-OAM use cases.

Continuous Verification (CV) is used today to check that the network's performance is satisfactory. A usual method of CV is to generate test packets and transport them over the network. These ping packets are used at the receiver's side to derive network performance characteristics. However, in practice CV is configured and rarely modified afterwards due to implementation limitations inherent in hardware. CV is done at a slow rate and CV protocols are typically not event triggered but run continuously. Dynamic SLAs are hard to support since the CV rate cannot be adjusted in response to new network conditions. Moreover, more complex CV tests cannot be used to provide better resolution where they are needed for error detection, since programmability is not available in the existing implementations.

Erlang has certain features that are interesting for supporting more flexible CV-OAM use cases. The most important of which is the ability to dynamically update the running code. This would enable the dynamic change of CV according to new SLAs. Furthermore, since Erlang is a programming language, it is possible to add CV routines where the CV rate is adjusted per the network conditions e.g., rates increasing when the network traffic load is increasing. Moreover, the Erlang VM's distributed capabilities are very useful for scaling the monitoring performance per connection link. Last, Erlang has been designed to make clustering of nodes as transparent as possible to the user/developer by managing node connectivity and an Erlang application's distributed processes addressing through its VM. Such capabilities allow for automatic node catalogue population, a problem that is important to Telcos since the vast size of their equipment and its interconnections a lot of times have to been

catalogued manually.

To investigate Erlang's capabilities a small experiment was setup with the option to be expanded if the results were promising. The experiment consisted of 3 Raspberry Pi devices connected through a switch in a star network topology. Each Raspberry Pi was operating Raspbian, a Linux based Operating System, and each had the latest Erlang language and Erlang VM installed on them. On each Raspberry Pi an Erlang node was started, and all 3 nodes were setup in a cluster using Erlang's built-in clustering capabilities.

At first a simple end-to-end PING engine for CV was created with each Erlang node hosting a user coded message sender and a message receiver function. Erlang allows the exchange of messages between functions using each function's Process ID (PID). When a cluster exists, where there is more than one node, PIDs are still used for the message exchanges. The Erlang VM manages the PID mapping between the different nodes. What was discovered was that although Erlang does provide PING functionality through *net_adm:ping(nameofnode)* it doesn't produce a roundtrip time. Wrapping net_adm:ping in a timer function, timer:tc(net_adm:ping(nameofnode)) did provide the roundtrip, but it was also measuring the code execution time of net_adm:ping. The solution for this was programmatic. New functions/processes were created in each node that allowed the message exchange between two nodes and measuring of the roundtrip time, but also the time each message took to traverse towards one direction of the link. Thus, greater resolution of the roundtrip was achieved. Extra code was added that depending on the variation between the send and return time, it was changing the rate of the messages exchanged while messaging the Erlang nodes that were involved about the rate change. Furthermore, a function/process called Messaging Queue that stores any message sent to it was created. Messages in the Message Queue were printed in the terminal. As was expected with Erlang being a programming language and with the built-in clustering capabilities it does provide CV flexibility. The next step was to include further network and node diagnostics e.g., physical node network ports load, thus further capabilities investigation was required.

The attempt to add more network and node diagnostics revealed that Erlang provides the following functions oriented for network packets creation, exchange, and Erlang cluster operation. At the same time, it provides no access to network hardware parameters or metrics and treats the physical network as transparent for the user and the developer, however both of these are important for a Telco/Network provider.

- **Net**: Network interface routines for packet creation and exchange.
- **Net_adm**: Limited inter-node administration routines, e.g. ping and hostname.
- **Net_kernel**: Kernel level registered process with routines for setting Erlang distributed operation
- **Gen_tcp**: Interface for the operation of TCP sockets.
- **Gen_udp**: Interface for the operation of UDP sockets.

- **Inet**: Provides access to TCP/IP protocols and routines like IP parsing, getifaddress, address and port number for sockets
- **Socket**: This module provides an API for network socket. Functions are provided to create, delete and manipulate the sockets as well as sending and receiving data on them.

On the side of node monitoring capabilities Erlang fares better, although Information Technology (IT) is still very limited. It offers the OS_Mon application, a built-in application which provides the following three processes.

- **Cpu_sup**: Supervises the CPU load and CPU utilization.
- **Disksup**: Supervises the available disk space in the system.
- **Memsup**: Supervises the memory usage for the system and for individual processes.

Despite OS_MON offering node performance information, there is a total lack of network monitoring functionality like traceroute or tcpdump, which is very important for a Telco/Network provider.

Erlang also offers two tools that could be used to access information from the host Operating System (OS). OS.CMD and erlang:open_port. Both functions can be used to execute host OS commands. An example code was written using both methods. It called the Linux *route -n* command, writing the output in a file and at the same time parsing the output within Erlang. The parser had to be coded from scratch.

In both cases the ease of use that the pre-existing Linux tools provide is very useful for providing the network information into the Erlang CV code, however, for every different Linux command a different parser must be created. Furthermore, every time the commands are updated the corresponding parsers must be updated too. Increasing the tool maintenance costs and increasing chances of system errors.

As part of the CV-OAM, when network conditions deteriorate, the deployed edge applications might need to be moved to another node. During the experiments Erlang capabilities like process links, monitors and supervisors were tested to investigate the resiliency of Erlang's processes. Motivated by Erlang's ability to failover processes to other nodes, running processes migration to other nodes was investigated. Initially, by calling manually the failover processes and as a second step by manually trying to change the failover node priority order. Both were not possible because the failover/takeover mechanism is setup in the configuration prior to starting the application and is immutable. Another approach was to separately spawn a process from one node to another node, however it also failed. That is due to the need during the spawning action to define a code module that contains the code of the migrating process on the new node and that essentially requires the codebase of the migrating code to pre-exist to the new node.

We investigated two other ways to achieve migration. The first is to create an Erlang code method where three things need to be considered:

- Process state
- Process registered name (PID)
- Messages Queue

The process state can be paused between messages sending and the PID can be registered in a central register allowing reuse in a new node. However, the message queue is trickier because messages can be lost if they are mid transition when migration happens, or when they are stored in the host OS of the node to be migrated.

To bypass these issues, the second method is to program these capabilities into Erlang by modifying the language itself, which of course can cause issues since maintenance will not be supported by the official version.

In the end Erlang has benefits but also weaknesses for the implementation of CV-OAM. Programmability is its biggest asset together with its clustering mechanism. It provides the capability to adapt according to SLAs and host performance or due to events. However, Erlang does not provide any network monitoring capabilities, but simply the most basic of ingredients that the network operator can use to monitor the status and performance of the network. Lastly, its migration capabilities of failover and takeover, do not leave much room for dynamic migration unless the applications themselves are written with migration in mind. Erlang despite its very promising capabilities, it is not a suitable tool for the implementation of CV-OAM for Telcos/Network Operators.

# 7 Conclusions

This deliverable has illustrated Piccolo's main architectural concepts with respect to distributed computing by describing different embodiments and corresponding experiments by the project.

Piccolo is addressing different use cases and environments, from a telco edge to factory floors. Whereas each of the corresponding developments follows the general Piccolo agent model, they provide different approaches to distributed computing.

## 7.1 Orchestration

In most mainstream distributed computing systems, resource management is addressed by orchestration systems such as Kubernetes. Whereas existing systems are typically developed for rather homogeneous environments, we have shown in Section 4.1 how Piccolo can provide semi-decentralized management by fragmenting edge infrastructure into multiple federated clusters. In section 4.2, we have described an approach to autonomous management and self-organizing capabilities for application services that can be implemented as an extension to existing virtualization technologies (e.g. virtual machines, containers).

The ICN-based Dataflow system introduced in Section 4.3 is an example of an inherently decentralized system that is aimed at managing resources without an explicit centralized orchestrator. From a Piccolo perspective, the Piccolo agents would coordinate directly, without necessarily requiring changes to local APIs to actors.

The Hybrid Orchestration approach described in section 4.4 employs a logically centralized coordinator that gathers information from compute nodes asynchronously, assisting the decentralized resolution mechanisms of the compute nodes. Note that in centralized orchestration solutions the forwarding nodes communicate with the orchestrator for every compute request resolution. By contrast, the hybrid orchestrator operates asynchronously to the existing resolution strategies by polling the network devices periodically on the control plane and providing optimization suggestions to nodes.

The Smart Factory application as illustrated in section 4.5 provides yet another environment that is characterized by specific requirements (reliable factory machine control in an edge-only network). Corresponding systems are modeled as IEC 61499 function blocks which are represented by Piccolo functions that need to be placed in a given compute infrastructure. This placement is primarily based on the location of actuators and sensors for certain functions but also on compute capabilities and topology considerations like node neighbourhood, employing location-independent naming/addressing. Optimizing the placement can be done by specific routing protocols or application-level extensions in the Erlang execution environments.

 

## 7.2 Resource Management

Piccolo has developed several approaches to resource management in distributed systems, and in this deliverable we have described how some of the Piccolo developments approach different aspects of resource management, such as compute resource allocation (for scaling etc.) and network resource management.

With respect to allocating compute resources, Piccolo has developed distributed approaches as the ICN-based Dataflow System (section 5.1) and the auction-based allocation system in the TINC platform (section 5.2).

With respect to network resource management, section 5.3 describes the resources that are relevant on one TSN-enabled node. In section 5.1, we have described the receiver-driven congestion control scheme in the ICN-based Dataflow system that enables joint optimization of compute and networking resources by adapting data interest rates to the performance of the compute function in a Dataflow node.

## 7.3 Next Steps

In this document we advance the work in various practical implementations, concentrating on two main topics: orchestration and resource management - with sub-sections about the detailed work for each of the proof of concept demonstrators. At this point we have taken a bottom-up approach, in order to get the demonstrators working. It also shows that the Piccolo mechanisms and APIs can be implemented on a variety of platforms and programming environments. One next step is to re-integrate the learnings from the implementations back into the more conceptually unified framework of the top-down architecture.

Existing frameworks make strong assumptions about the underlying infrastructure for optimal operation, which was found to be a limiting factor when porting to edge infrastructures [8]. For example, most frameworks requires all processing resources to be in the same cluster (and therefore directly reachable from each other), just like data centres, which does not always hold true for processing units in the edge. The most inhibiting factors affecting the performance of existing orchestration frameworks in edge environments are: (i) the ability to handle the heterogeneity of the devices making up the infrastructure (ii) the scale of devices that span large geographical regions and (iii) different management entities that may collaborate together to contribute their resources to a larger infrastructure. Our on-going work aims to alleviate these inhibiting factors.

The next steps for the various activities described in this document include the following:

**Next Steps in TINC's resource allocation through auctions**: We aim to extend the security features of the auction-based mechanism by identifying nodes that act untruthfully by placing bids for resources that are not equal to their actual gain in terms of QoS in order to disrupt the market's

equilibrium. Effectively we will develop mechanisms that define strict policies of bid placement and perform decrease of a node's reputation.

**Smart Factory next steps**: As explained in Del2.2 [1] implementation of the Piccolo node for a Smart Factory has not started yet. Hence aspects of distribution explained in this document are also part of implementation efforts over the next months. In terms of concepts, the agent leader election is not very mature at this stage and will certainly to be revised. By far the greatest challenge is the idea of a unified Piccolo protocol which reflects on topology observation aspects as well as on function placement, and maybe even routing of compute as described by other use cases.

**Next Steps in ICN-based Dataflow System**: We plan to make the system adaptive to the deployed application, i.e., optimize the system's configurations according to the use case. Moreover, as scaling decisions are handled in a decentralized approach, we aim to enhance it by considering more metrics and evaluating the effect of these decisions on the whole system. Also, further work will be done on the resource allocation and actor placement when recovering from failure or scaling an actor.

**Next Steps for Telco Edge Cloud**: GSMA is currently discussing how it works with open source and developer communities, such as Linux Foundation, and what its role should be. The aim is for the technical and commercial realisation of a federation of operators' edge clouds. We will continue to contribute to this activity.

**Next Steps for CV-OAM**: The research on CV-OAM for Telcos will look into eBPF and how can it be used to extend monitoring capabilities for different SLAs, using its inherent ability to extend kernel code without changing the kernel source code or loading kernel modules.

# References


[1] Piccolo Project. *Initial Report on PoC Implementation of a Piccolo Node*. Tech. rep. Deliverable D2.2. 2021.

[2] Piccolo Project. *Application Design and Development Report*. Tech. rep. Deliverable D1.2. 2021.

[3] Piccolo Project. *Architectural invariants for distributed computing and technical requirements*. Tech. rep. Deliverable D3.1. 2021. URL: https://www.piccolo-project.org/assets/Deliverables/Piccolo_Del3.1_Architectural_invariants_for_distributed_computing_and_technical_requirements.pdf.

[4] Piccolo Project. *Use Cases, Application Designs and Technical Requirements*. Tech. rep. Deliverable D1.1. 2021. URL: https://piccolo-project.org/assets/Deliverables/Piccolo_Del1.1_Use_cases__application_designs_and_technical_requirements.pdf.

[5] Piccolo Project. *Piccolo Node definition*. Tech. rep. Deliverable D2.1. 2021. URL: https://www.piccolo-project.org/assets/Deliverables/Piccolo_Del2.1_Piccolo_Node_definition.pdf.

[6] Alejandro Cartas et al. "A reality check on inference at mobile networks edge". In: *2nd Workshop on Edge Systems (EdgeSys)*. 2019.

[7] Nitinder Mohan and Jussi Kangasharju. "Edge-Fog cloud: A distributed cloud for Internet of Things computations". In: *2016 Cloudification of the Internet of Things (CIoT)*. IEEE. 2016, pp. 1–6.

[8] Andrew Jeffery, Heidi Howard, and Richard Mortier. "Rearchitecting Kubernetes for the Edge". In: *4th ACM EdgeSys* (2021).

[9] Aleksandr Zavodovski et al. "ExEC: Elastic extensible edge cloud". In: *2nd Workshop on Edge Systems (EdgeSys)*. 2019.

[10] Nitinder Mohan and Jussi Kangasharju. "Placing it right!: optimizing energy, processing, and transport in Edge-Fog clouds". In: *Annals of Telecommunications* (2018).

[11] Aleksandr Zavodovski et al. "Icon: Intelligent container overlays". In: *17th ACM Workshop on Hot Topics in Networks (HotNets)*. 2018.

[12] L. Zhang et al. "Named Data Networking". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 66–73. ISSN: 0146-4833. DOI: 10.1145/2656877.2656887. URL: https://doi.org/10.1145/2656877.2656887.

[13] *Apache Flink – Stateful Computations over Data Streams*. The Apache Software Foundation. https://flink.apache.org/.

[14] *Google Dataflow*. Google. https://cloud.google.com/dataflow.

[15] *Flink Architecture*. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/flink-architecture/. Accessed: 2021-09-09.

[16] Dirk Kutscher, Laura Al Wardani, and T M Rayhan Gias. "Vision: Information-Centric Dataflow – Re-Imagining Reactive Distributed Computing". In: (2021). DOI: 10.1145/3460417.3482975. URL: https://doi.org/10.1145/3460417.3482975.


 

[17] 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.

[18] Van Jacobson et al. “Networking Named Content”. In: *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’09. Rome, Italy: Association for Computing Machinery, 2009, pp. 1–12. ISBN: 9781605586366. DOI: 10.1145/1658939.1658941. URL: https://doi.org/10.1145/1658939.1658941.

[19] 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.

[20] Michał Król and Ioannis Psaras. “NFaaS: Named Function as a Service”. In: *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ICN ’17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 134–144. ISBN: 9781450351225. DOI: 10.1145/3125719.3125727. URL: https://doi.org/10.1145/3125719.3125727.

[21] Christopher Scherb et al. “Resolution strategies for networking the IoT at the edge via named functions”. In: *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*. 2018, pp. 1–6. DOI: 10.1109/CCNC.2018.8319235.

[22] Marica Amadeo et al. “SDN-Managed Provisioning of Named Computing Services in Edge Infrastructures”. In: *IEEE Transactions on Network and Service Management* 16.4 (2019), pp. 1464–1478. DOI: 10.1109/TNSM.2019.2945497.

[23] A K M Mahmudul Hoque et al. “NLSR: Named-Data Link State Routing Protocol”. In: *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking*. ICN ’13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 15–20. ISBN: 9781450321792. DOI: 10.1145/2491224.2491231. URL: https://doi.org/10.1145/2491224.2491231.

[24] *Erlang/OTP*. https://erlang.org/doc/index.html. Accessed: 2021-09-13.

[25] *Mnesia User’s Guide*. https://erlang.org/doc/apps/mnesia/users_guide.html. Accessed: 2021-09-13.

[26] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016).

[27] Dimitris Vyzovitis et al. “Gossipsub-v1.1 Evaluation Report”. In: (2020). URL: https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md.

[28] 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.

[29] 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.

[30] Dinh Ngoc Tu et al. “Everything You Should Know about Intel SGX Performance on Virtualized Systems”. In: 2021.

 

[31] Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. “On the Evolution of ndnSIM: an Open-Source Simulator for NDN Experimentation”. In: *ACM Computer Communication Review* (July 2017).

[32] George F. Riley and Thomas R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation*. Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. ISBN: 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_2. URL: https://doi.org/10.1007/978-3-642-12331-3_2.

[33] Christian Tschudin and Manolis Sifalakis. “Named functions and cached computations”. In: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. 2014, pp. 851–857. DOI: 10.1109/CCNC.2014.6940518.

[34] 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.

[35] *5G Operator Platform*. https://www.gsma.com/futurenetworks/5g-operator-platform/. Accessed: 2021-09-13.

 