PICCOLO

CELTIC-NEXT

Deliverable D2.2

Initial Report on PoC Implementation of a Piccolo Node

Editor:	Marian Ulbricht – InnoRoute
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public
Contractual delivery date:	September 30, 2021
Actual delivery date:	September 30, 2021
Suggested readers:	Researchers, developers and technologists interested in edge computing and the convergence of networking and computing.
Version:	1.0
Total number of pages:	43
Keywords:	Architecture, Distributed Computing, Protocols, Node

Abstract

This document describes the state of the Piccolo node Proof of Concept implementations, at the halfway point of the project.

Disclaimer

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

Impressum

[Full project title] Piccolo: In-Network Compute for 5G Services
[Short project title] PICCOLO
[Number and title of work-package] WP2 Piccolo Node
[Number and title of task] T2.2 Piccolo Node development
[Document title] D2.2 Initial Report on PoC Implementation of a Piccolo Node
[Editor: Name, company] Marian Ulbricht, InnoRoute
[Work-package leader: Name, company] Chris Adeniyi-Jones, ARM

Copyright notice

Executive Summary

The Piccolo Node is an individual node in a Piccolo network. It delivers an open, low latency, efficient, secure, in-network compute implementation.

This deliverable D2.2 "Initial Report on PoC Implementation of a Piccolo Node" describes the current state of the various Proof of Concept (PoC) implementations of Piccolo nodes.

List of Authors

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

Table of Contents

List of Figures

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
BEAM	Bogdan's (or Björn's) Erlang Abstract Machine
CPU	Central Processing Unit
DHT	Distributed Hash Table
EE	Execution Environment
FPGA	Field Programmable Gate Array
I4.0	Industry 4.0
JSON	JavaScript Object Notation
LWVM	Lightweight Virtual Machine
MMI	Memory Mapped Interface
NUC	Next Unit of Computing
OPC	Open Platform Communications
OPCUA	OPC Unified Architecture
OTP	Open Telecom Platform
PoC	Proof of Concept
REST	Representational State Transfer
RPC	Remote Procedure Call
TAS	Time Aware Shaper
TEE	Trusted Execution Environment
TINC	Trusted In-Network Computing
TLS	Transport Level Security
TSN	Time-Sensitive Networking
UI	User Interface
VM	Virtual Machine
VPE	Visual Processing Engine
WASM	WebAssembly

Definitions

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

TrustNode: Hardware-accelerated routing devices that include a Field Programmable Gate Array (FPGA) for fast routing and a supporting Central Processing Unit (CPU) for advanced packet processing. For more information, please refer to http://TrustNo.de.

RaspberryPI: Well-known singleboard computer for rapid prototyping and research. Designed in the UK. For more information, please refer to https://www.raspberrypi.org/.

RealTimePI: RaspberryPI with additional RealTimeHAT (Hardware Attached on Top) to enable hardware accelerated Time-Sensitive Networking (TSN) and time synchronisation features on the RaspberryPI. For more information, please refer to https://innoroute.com/realtimehat/.

ESP32: Microcontroller with built-in Wi-Fi connectivity manufactured by Espressif Systems. For more information, please refer to https://www.espressif.com/en/products/socs/esp32.

1 Introduction

This document captures the current status of our work on Piccolo nodes, that is the individual nodes in an In-Network Computing system.

We are developing various implementations of a Piccolo node, in order to address the various needs of the different Proof of Concept (PoC) demonstrators and use cases. This bottom-up approach enables us to make progress, get the PoC demos working, and give us the opportunity to learn-by-doing.

At this intermediate point within the project, the 4 different PoC Piccolo nodes described in this document do not yet fully match the concepts of Del2.1 [1]. That document took a top-down approach and defined a high-level node architecture and interfaces. Future work will examine the compromises taken so far in order to accelerate implementation, and update the architecture in the light of the practical lessons. In this document, each node description contains adapted versions of the architecture and interface figures of Del2.1 [1]. The Conclusions chapter gives some challenges for each Piccolo node implementation, together with directions for further development.

2 Piccolo node general view

In section 5 of the first deliverable on the Piccolo node (D2.1) [1], we set out an initial node architecture for a Piccolo node which could embrace a range of environments for hosting Piccolo functions. The primary reference diagram is reproduced again in Figure 1.

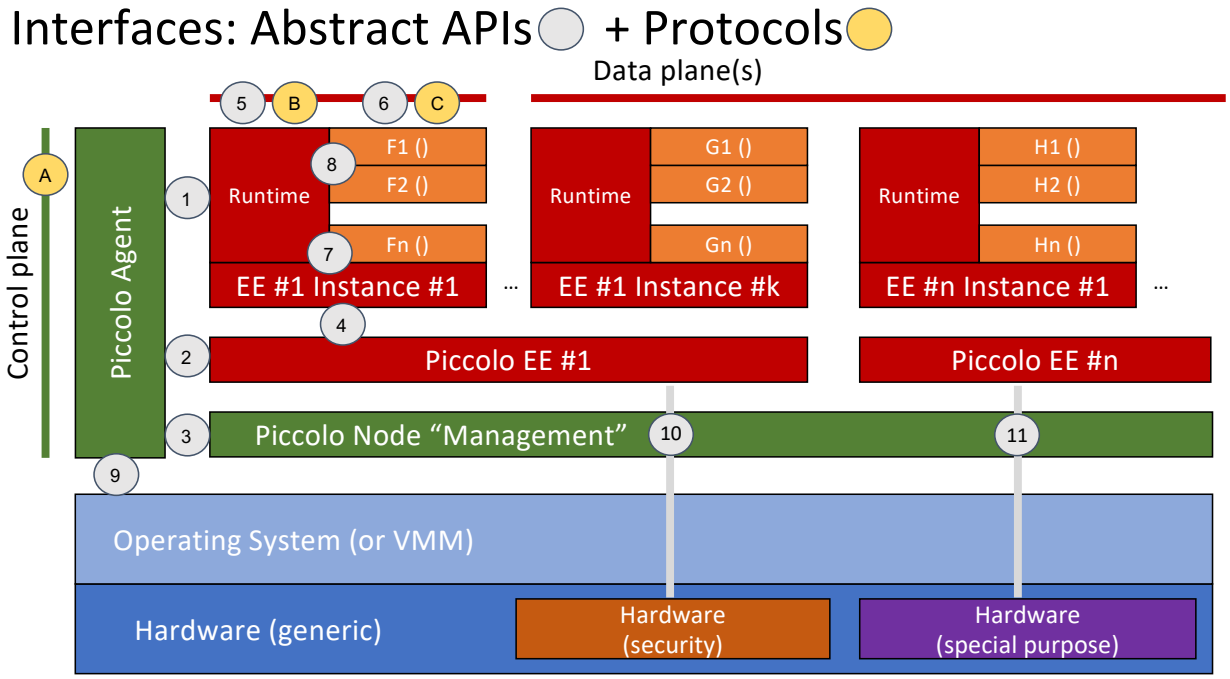


Figure 1: Abstract API and Protocols (note: EE = Execution Environment ; VMM = Virtual Machine Manager ; F1, G1, H1 etc are different functions)

In this next stage of the project, a number of PoCs have been developed, each one focused on a specific use case. An important objective of these PoCs is to understand efficient implementation for each one **without any strong "top-down" architectural constraints**. We hope this 'bottom-up' development will make it possible to gain important insights which would not necessarily be apparent prior to developing each PoC. Also, for the overall architecture to be developed in such a way that it is fully compatible with a wide variety of specific implementation scenarios and does not place unforeseen and unnecessary constraints on specific implementations.

As these PoCs mature the learning from them will be understood and taken into account. Over the coming months the project will develop a more complete overall architecture and specification. This report is therefore focused on the details of each PoC and does not, at this stage, consider how the significant implementation differences between them will be integrated into a single coherent and detailed architecture; that will come later in the project.

A particularly important feature of the node architecture is the Piccolo node's Application Programming Interface (API) (protocol A) in Figure 1. It is already evident that it is unlikely that any API

definition developed by Piccolo will be immediately adopted and implemented directly by each and every hosting and execution environment over which Piccolo has little influence. This suggests that Piccolo should adopt a "plug-in" architecture to support the Piccolo node API as outlined below.

A second significant feature to emerge from the layered hierarchy inherent in the initial node architecture of Figure 1 is that there are many different contexts and each context may have different users. This suggests that the Piccolo node API needs to support access control to different users according to context.

2.1 Piccolo Agent as a plug-in architecture

In order to interact with the APIs presented by existing Execution Environments (EEs), some of which are used in the PoCs described in Section 3 below, the Piccolo agent can act as an interface translator between a single common Piccolo node API and these EE specific APIs. This is illustrated in Figure 2.

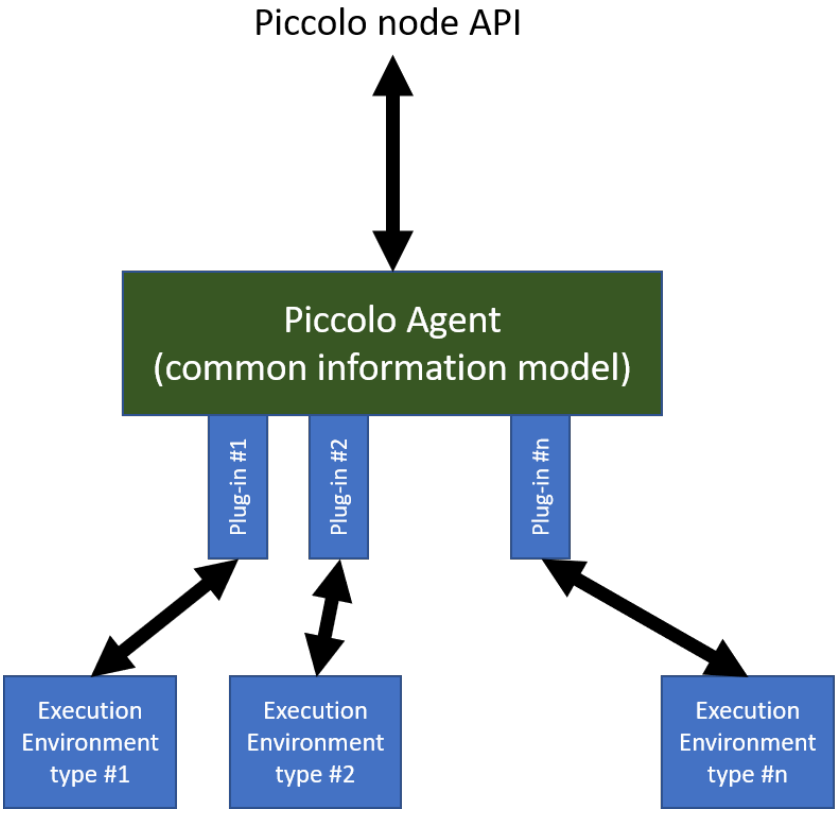


Figure 2: Plugin model for the Piccolo node API

Given that there are a range of different types of EEs with different APIs, an efficient way of implementing translation is to use a plug-in architecture. In this case, the Piccolo agent has a common information model which is consistent with the Piccolo node API. When there is a requirement to interface to a specific EE API, a specific module of code can be written which can turn protocol transactions on the Piccolo node API into protocol transactions on the EE specific API. This translation

can make use of the common information model both as a means of correctly translating parameters which may have different syntax and also as a means of holding any state if this is necessary to handle any differences in protocol message sequencing.

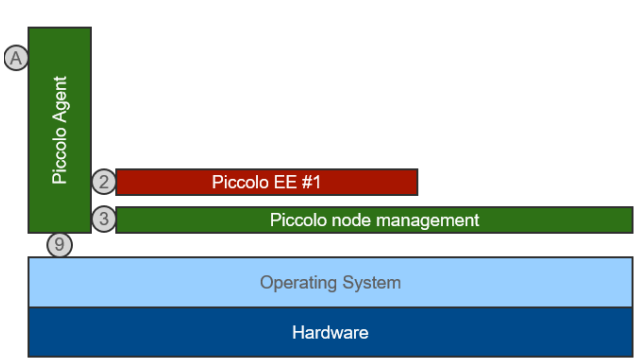
It may also be necessary to adjust for information difference between the Piccolo node API and the EE specific API. For example, as the Piccolo node API is likely to be more abstract than the EE specific API, the EE specific API may require more parameters than can be supplied by the Piccolo node API. In this case, the Piccolo agent can:

- fill in default values;
- use a policy to calculate values based on other information held by the Piccolo agent;
- provide a pass through so that the user can set the values (this solution is less ideal as the Piccolo node API is now no longer a sufficient API).

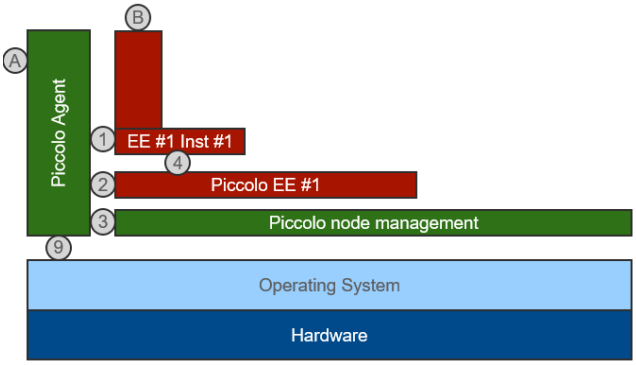
It is possible, if less likely that the Piccolo API allows for a greater level of configuration than is supported by the specific EE API. In this case, the Piccolo agent plugin can translate the requirement to whatever more granular EE specific configuration can satisfy the request. It is also possible with this plugin architecture to physically separate the Piccolo agent from the execution hardware which may be an advantage for some very lightweight execution hardware.

2.2 Multi-Context Nature of the Piccolo API

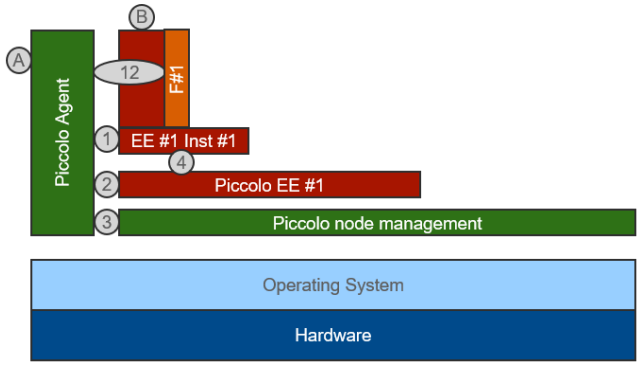
A second aspect to Piccolo agent which will be addressed when the more detailed work on developing the Piccolo node API is undertaken in the coming months is a security model which includes context. How different contexts arise is illustrated in Figure 3 which shows four different contexts as different layers of software are added. The ownership at each stage may be different and it may be important, for example, that owners at higher layers are not allowed access to configuration information and control at lower layers.



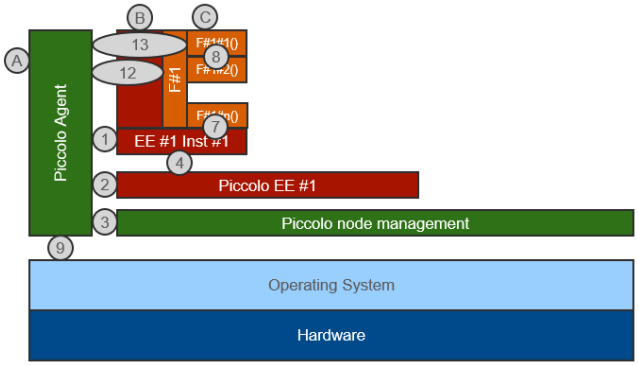
((a)) Context 1 – Create EE type #1



((b)) Context 2 – Create EE#1 instance #1



((c)) Context 3 – Create function type #1



((d)) Context 4 – Create function #1 instance #1

Figure 3: Four example contexts in the use of the Piccolo node API

3 Piccolo node implementations

3.1 Behavioural Risk

The behavioral risk monitoring PoC, as described in D1.1 and D1.2 [2, 3], deals with the computation and evaluation of a risk index for a driving trip. For this PoC, we install interior cameras within the cockpit of the vehicle, being able to observe the driver physical behaviour and reactions. Additionally, in-vehicular data (e.g. acceleration, velocity, steering angle etc.) is available to be shared with the camera data to capture how the vehicle is being driven. Together the captured data (vision along with in-vehicular data) is processed in a data processing pipeline implementing a behavioral risk evaluation model. The implemented data processing pipeline functionally consists of two parts: (i) the vision processing pipeline, and (ii) the in-vehicular data processing pipeline (cf. Figure 4).

In the context of this document, the node specific compute aspects required to support the data processing pipeline are of interest. As presented in Piccolo D1.2 [3], there are limiting factors to execute compute intensive operations within the vehicle. As an example of a limiting factor, in-vehicle computing platforms like vehicle computers are restricted regarding their compute capabilities and have to be used judiciously, and therefore, the execution of the entire pipeline (cf. Figure 4) within the vehicle is infeasible. Hence, the components of the pipeline within the vehicle are limited to only those which are necessary for data acquisition. For example: the processing of camera stream to generate real-time video streams for further processing; the capture of data from in-vehicle field buses such as Controller Area Networks to make them available as data streams via the OBD-II interface. In order to handle the compute intensive operations of processing and augmenting the different data streams, the business logic implementing the behavioural risk monitoring is hosted within the Piccolo infrastructure.

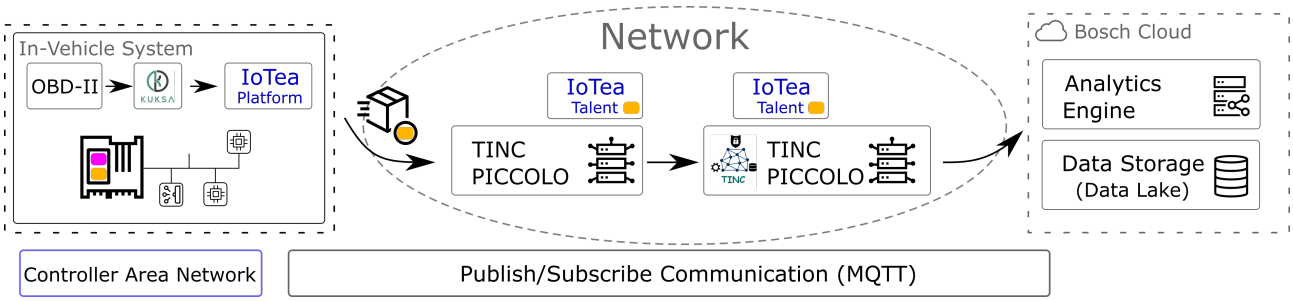


Figure 4: Vehicle Data Processing Pipeline for Behavioral Risk Monitoring as an example of one out of two different processing pipelines to be executed on Piccolo nodes.

In order to handle the compute limitations mentioned, e.g., handle complex workflow definitions for *IoTea* Talents, dynamically offloading the compute load to Piccolo nodes within the infrastructure is desirable. In the Risk Behavioral Monitoring prototype, we are using the Trusted In-Network Computing (TINC) Piccolo node for offloading functions. The platform allows for flexible deployment of services and functions and offers the option to either execute them in a secure or non-secure fashion dependent on the business goal of the service/function provider. For example, in the context of

the vehicle processing pipeline, *IoTea* Talents implementing business logic to fuse data from several vehicles might be executed in a secure fashion, while other talents might not have this requirement, being executed in a normal fashion.

The following subsection provides an overview of the TINC platform, representing a Piccolo Node, and illustrate design principles and the implemented Piccolo APIs.

3.1.1 Trusted In-Network Computing Node

The Trusted In-Network Computing node, is a Golang implementation of the abstract Piccolo Node and offers a platform to execute functions in the form of Docker containers in a both a normal and a secure manner by utilizing Trusted EEs [4]. The TINC implementation is used in the Behavioural Risk Monitoring PoC as a basis for compute nodes available at the edge of a communication infrastructure. It is based upon the concepts and API specifications described in D2.1 [1]. The TINC node is a x86 device that uses Linux as an operating system, and additionally provides Intel Software Guard eXtensions (SGX) [5] support implemented as part of the TINC-engine. The TINC-engine itself is an implementation of the conceptual Piccolo Agent [1]. For the PoC we use an Intel Next Unit of Computing (NUC) Kit (cf. Figure 5) [2]. However, any node that has the aforementioned characteristics such as supporting Intel SGX [5] can be used. Figure 6 illustrates a high-level overview of a TINC/Piccolo node implemented based on the Figure 11 in Piccolo D2.1 [1].

Figure 5: Intel NUC as a Piccolo Node enabler

Piccolo functions used in TINC are realized as Docker containers with the additional ability to provide computations in Trusted Execution Environments (TEEs) [4] such as Intel SGX [5]. The added value of such a property is that each function can run inside an enclave and be isolated from any other

[1] On this section the terms TINC-engine and Piccolo agent will be used interchangeably.
[2] The models that will be used are: NUC7PJYH and NUC7I3DNKE

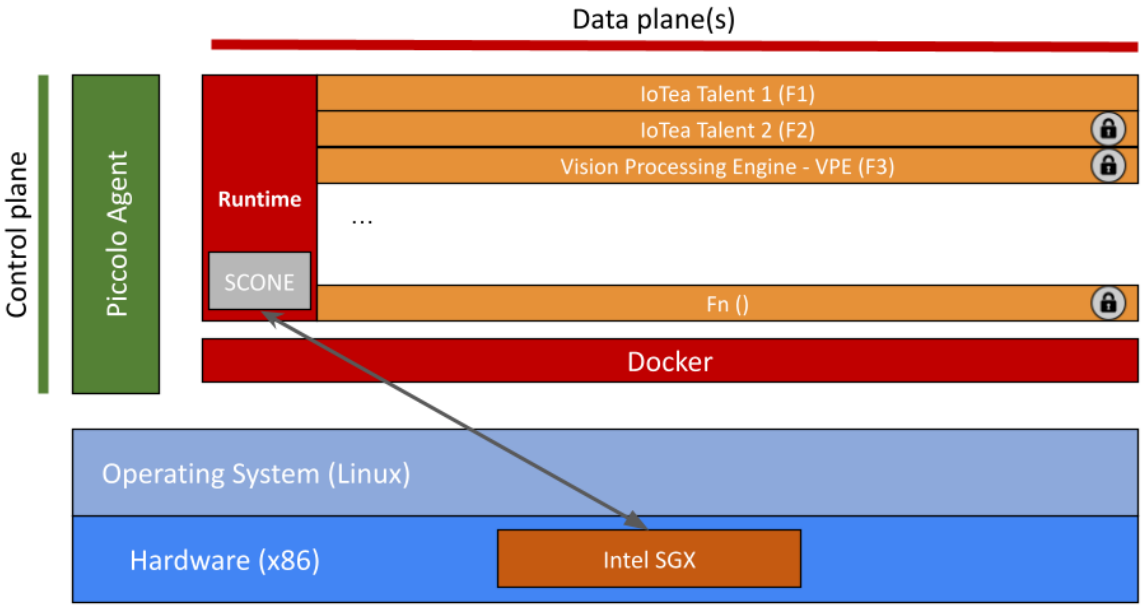


Figure 6: TINC as a Piccolo Node. Adapted from D2.1 [1]

function running on the same Docker container, on another container executed at the same time, or any privileged software running in the host OS. Such a feature, along with a virtualization environment like Docker, enables support for *multi-tenancy* since the function execution and communication is isolated from other tenant operations (cf. Figure 6, presented with a shielded lock icon). Besides the multi-tenancy feature, the isolation also protects against interference of sensitive operations. To provide seamless support between Docker containers and the Intel SGX technology, we leverage from the Secure Container (SCONE) Framework [6] in order to protect computations taking place inside containers and additionally offer attestation procedures to verify the freshness and correctness of the code that will run on the platform.

In the TINC platform, a Piccolo function is a containerized confidential function that has been generated using a utility called TINCmate. This utility converts a native application to a (secure) Piccolo function and ultimately generates a Docker image that is ready to be loaded on a TINC/Piccolo Node. A converted Piccolo function implements (but not mandatory) its own HTTP server in order to be reachable from external parties that want to trigger actions related to the function.

As the concept supports theoretically any type of containerized function, the current implementation of the secure/isolated execution supports currently programming languages such as Python, Nodejs, Golang, C/C++ and frameworks such as Tensorflow [7] and Openvino [8]. The secure variants in the Behavioral Risk Monitoring prototype[3] are provisioned by Fluentic and ready to be deployed as Piccolo function on TINC.

[3] A secure variant is a native application that has been re-compiled (or linked against a particular linker provided by SCONE) using a set of cross-compilers and is able to be executed inside an Intel SGX enclave or an interpreter that runs inside an enclave

A Manifest describes a function description (before its execution), serves as the entry point for the function deployment/management logic and is used to plan resource allocation and function/data orchestration[4]. It contains properties like execution arguments, environment-related variables and also TEE specific requirements. Moreover, it can describe a set of functions and their properties, grouped together to define a compute workload. The introduction of such Manifests in TINC further supports the identification of the most suitable node for execution by the TINC/Piccolo agent and enables the definition of access control and access operation rules between functions. A TINC Manifest is provided in the form of a JavaScript Object Notation (JSON) file. Listing 1 gives an example for a *IoTea* talent function. The structure begins with the definition of a workload that is used to group functions together (under one namespace). Then each function's properties or requirements (e.g. process as a secure container) are defined in order to be interpreted by a TINC/Piccolo node. Functions which are not of the same compute workload definition are isolated from each other, not able to share any data between each other.

```
{
    "api_version":"v0.0.3"
        "workload": {
        "workloadID": "3dec562b18a9050f0b5fbe0036cd7c9a9a3c3ebf59d3e5f323fd0995d
    7a79738",
        "name": "w1",
        "version": "v0.0.1",
        "description": "A basic python IoTea talent",
        "namespace": "piccolo-slice-1",
        "network": "iotea-platform-network",
        "functions": [
            {
                "functionID": "c91b4b74bf784a319bd046bf161edb1a",
                "name": "f1-iotea-talent",
                "kind": "secure:docker:container",
                "version": "v0.0.1",
                "compression": "tar.gz",
                "programming_language": "Python",
                "size": "390334976",
                "remote_reference": {
                  "download_endpoints": [
                    "https://tincmate.net:8080/api/v1/function/download/c91b4b74
    bf784a319bd046bf161edb1a/f1-iotea-talent.tar.gz"
                  ]
                },
                "processing": {
                   "secure_compute": {
                        "mode": "hw",
                        "metadata": {
                            "session_id": "piccolo-slice-1/f1-iotea-talent_a4a0b
    3",
                            "mr_enclave": "3430b3c0ab0e8a24ea4481e6022704cdbbcff
    68f6457eb0cddaecfd734fed241",
                            "class_size": "L"
```

[4]More details about resource allocation in TINC is described in Piccolo D3.2 [9]

```
                        }
                    }
                },
                "restart_on_error": true,
            }
        ]
    }
}
```

Listing 1: Example function deployment in TINC/Piccolo Node

The TINC node implements the basic operations defined by the Piccolo API (cf. Piccolo D2.1 [1]) including the discovery of other Piccolo nodes, the execution of Piccolo functions to remote nodes, the startup of functions in the execution environment and the query of function identifiers in the network. In more detail, the TINC/Piccolo node uses rendezvous points and Distributed Hash Tables (DHTs) for discovering other nodes while for obtaining information from compute nodes such as hosted functions or utilized resources, a publish/subscribe communication model is used. This loosely coupled model allows a Piccolo agent to operate in a *location-independent* as nodes are concerned about the data in the network and do not want to maintain locations of node to request for such information.

The TINC/Piccolo node utilizes Docker as the underlying layer to start/stop/remove/query functions in the form of (secure) Docker containers. Specifically, the runtime provides a watchdog called "Observer" that runs alongside each function[5] and aids its interactions not only with external entities but also with other functions. Additionally, the watchdog monitors the resource consumption and liveness of the function and reports it to the Piccolo Agent. The aforementioned description refers to the implementation of the **EE/Runtime-function** API. The described operations relate to functions that are already located in a TINC/Piccolo node. However, when that is not the case, the node leverages from the manifest files and the topic-based messaging patterns to identify and fetch functions from other nearby nodes, ultimately providing an implementation of the **Runtime-Network** API.

While the information exchange between nodes are more extensively implemented in the current node prototype, the interactions between functions (**Function-network API**) are limited in the sense that a function can communicate with another function only using their custom protocol or by leveraging an HTTP API provided by the watchdog that forwards requests to the relevant function without the requestor knowing the actual location of the target function but only using a function identifier provided by the watchdog. However, since this type of node allows also the execution of secure functions the communication is limited under functions that belong on the same workload and that run inside SGX enclaves. The policies for the interactions between functions are provided by a combination of two description documents. The first one is the manifest that presents which functions belong under the same workload (ultimately belong to the same tenant). The second one is a session policy that is generated from TINCmate and implements SCONE's specification in order to define the basic rules of communication between two functions in terms of enclave interaction. The latter is described more in Section 5.2. Due to session policies and manifests the data sharing between functions (**Runtime-**

[5]Note that in case of a secure function a secure watchdog is used that runs inside an Intel SGX enclave.

function-function API) is more controlled and governed through strict policies in case of sensitive functions (**Function protocols**).

Since this type of Piccolo node was derived from the TINC system certain APIs like the **Agent-Runtime**, **Agent-Execution environment**, **Execution environment to Execution Environment instance (plus runtime)** and **Agent-environment** are implemented by internal components of TINC. These components maintain a list of active/stopped functions and can at any time configure new slots for incoming ones[6] by contacting the docker daemon (i.e. bootstrapping a new container), increase the number of replicas, register a stored function for discovery through the publish/subscribe messaging system and finally remove unused functions. Removing functions is a combined mechanism based on predefined rules (setup by the node owner in a configuration file) and consumption metrics provided by the function watchdog.

As already highlighted this type of node provides support for sensitive computations, like functions operating on human traits and personal data (Behavioural Risk Monitoring PoC), through the utilization of SCONE, Intel SGX, Docker. The interface for accessing secure hardware like Intel SGX (**Agent/node management - secure hardware**) is implemented by SCONE and its basic operations have been consumed and simplified by TINC utilities (e.g. TINCmate). This combination allows not only to protect data at use, in transit and at rest but also to maintain a continuous data processing pipeline as described in the next section.

3.1.2 Integration with the Behavioural Risk PoC

Since the Behavioural Risk PoC requires access to personal data such as a person's facial expressions, body movements and sensitive vehicle metadata there is an inherent need for protecting this data along with the application code that is provided by the application/service providers like Sensing Feeling and Bosch. For instance, as it was presented in Figure 6 Bosch uses *IoTea* Talents (cf. Piccolo D1.2 [1]) as functions to be executed on top of a TINC/Piccolo node in a secure manner. Talents are based on a simplified programming actor-based model to describe the processing rules. In order to be executed in a secure fashion, IoTea Talents need to be converted to a secure variant of a Piccolo function. In Figure 7, we present the lifecycle of a Talent deployment in TINC/Piccolo Agent by utilizing TINC's utilities (TINCmate).

The capability to convert a native talent into a secure TINC/Piccolo function is called TINCmate and is a web service that provides a Representational State Transfer (REST) API along with a User Interface (UI). Note that all the operations until the talent is deployed into a TINC/Piccolo node take place into the TINCmate service.

The first step for the deployment on a TINC/Piccolo Node is to compress a talent[7]. Secondly, we use an operation called "Tincify"[8] that relieves the developer from configuring/converting the application

[6]The resource allocation is a more complex operation that is described in D3.2 [9].

[7]Compression formats supported are tar or gzip.

[8]Tincify: This procedure is based on SCONE's "sconify" [10] mechanism that generates a confidential docker image from

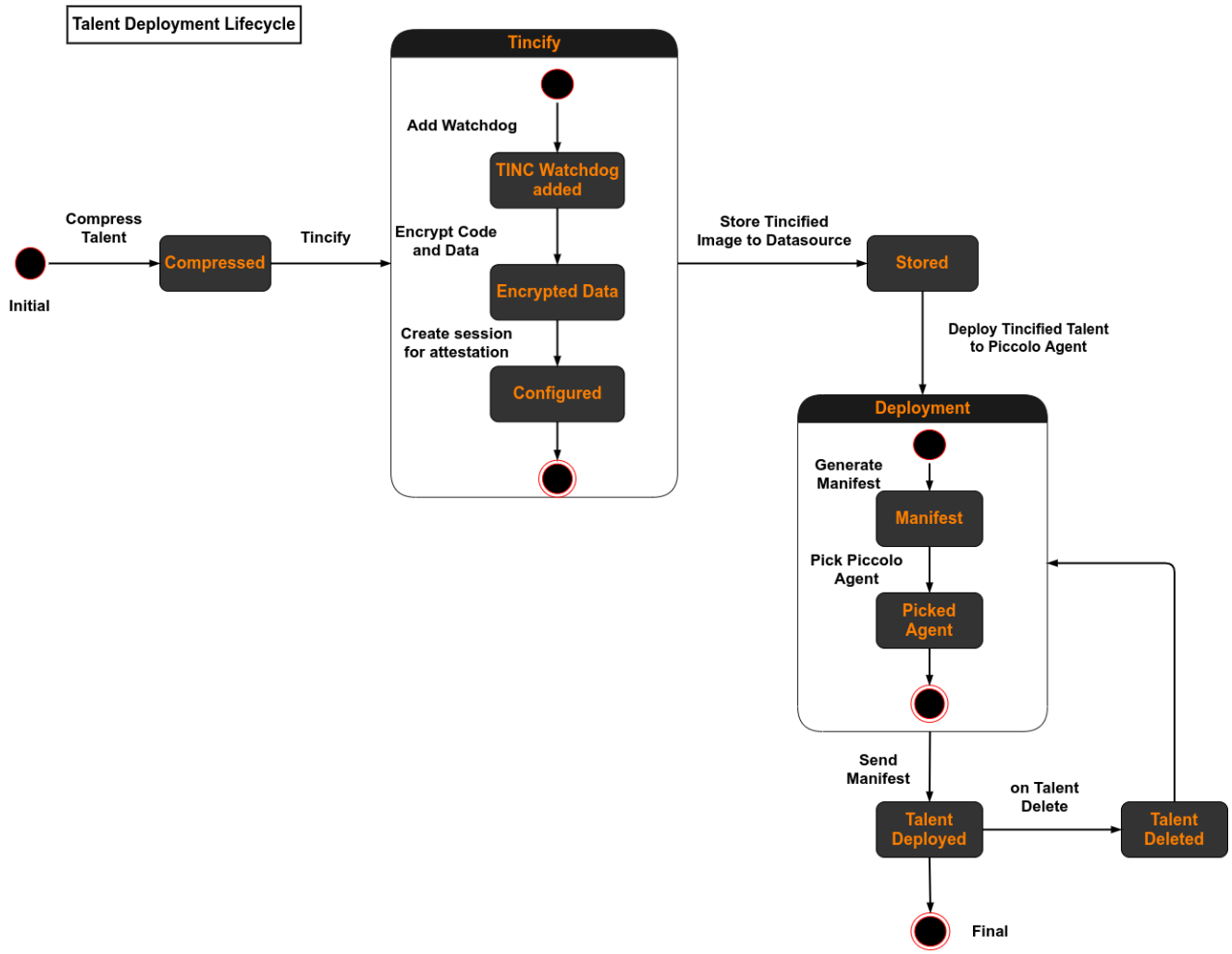


Figure 7: State machine diagram presenting the lifecycle of a talent's deployment on a TINC/Piccolo Node

into a secure TINC/Piccolo function. This mechanism receives as input compressed applications developed either in Python or Nodejs along with any required files or libraries needed for the execution of the application. The application will be containerized (added to a Docker Image as a layer) by using a specific template provided by Fluentic. This template includes a component called "Observer" that acts as a watchdog for the application/function and special purposed compilers and utilities from SCONE that enable the interaction with the Intel SGX driver in order to seamlessly convert it to a secure variant and make it compatible to run in a secure enclave [9].

The procedure handles sensitive files by encrypting them into secure regions. The procedure involves the creation of an index file that points to the structured file-system of the application. The final encryption keys are considered "secrets" and need to remain hidden from unauthorized entities. The procedure leverages from an external entity that offers attestation and session management operations (see details in Section 4.2). Once the session has been generated the "Tincify" procedure finishes by storing the produced docker image to the disk and makes it available through a REST API for

a native one but with the addition of a watchdog component and extra configurations by Fluentic Networks.
[9] enclave: private predefined areas in memory that shield code and data during computation

download upon request.

For the deployment state a manifest needs to be generated and a designated Piccolo Node chosen. Once the manifest is received by a TINC/Piccolo node and the function placement procedure takes place then the talent is considered deployed. To complete the lifecycle, a talent can be removed when it is considered unused (i.e. no requests received or is inactive) or on demand by the application provider. The same lifecycle also applies to the deployment of a Visual Processing Engine (VPE) which is the other data processing pipeline considered in this PoC.

3.2 Smart Factory

For the Smart Factory use case there is no PoC implementation of a Piccolo node yet. Instead efforts are focused on a simulator application which allows the participants to follow a top down approach to understand the overall system and related problems better. This simulator is described in D1.2 [3]. Hence the following description of a PoC node is of theoretical nature.

3.2.1 Node design

The Smart Factory use case promotes the idea of injecting and processing distributed applications based on IEC 61499 [11] into complex network topologies which control conveyor belt installations involving sensors and actuators. The value of this approach stems from far shorter (re-)configuration periods after a conveyor belt setup is changed and also an optimized control of the overall work piece flow. Hence the Smart Factory vision of a Piccolo node derives from:

- **Optimal placement of IEC 61499 function blocks (Piccolo functions) within a network topology**
- **Optimal Control of work piece flow leveraged by a digital twin of the conveyor belt network**

Both points require the exchange of information between nodes of the network topology which is dealt with in D3.2 [9]. However, the Piccolo node itself needs to maintain an information database which stores exchanged information. Further the information needs to be accessible to the optimization layers and in particular by the digital twin. It is likely that every node needs to be capable of topology-wide optimization for function placement since IEC 61499 applications can be injected on any node.

From this perspective the role of the Piccolo Agent within a node gets more precise:

- Maintain an information database through means of communication with other agents
- Enforce placement of IEC 61499 function blocks (Piccolo functions) as requested by optimization layers

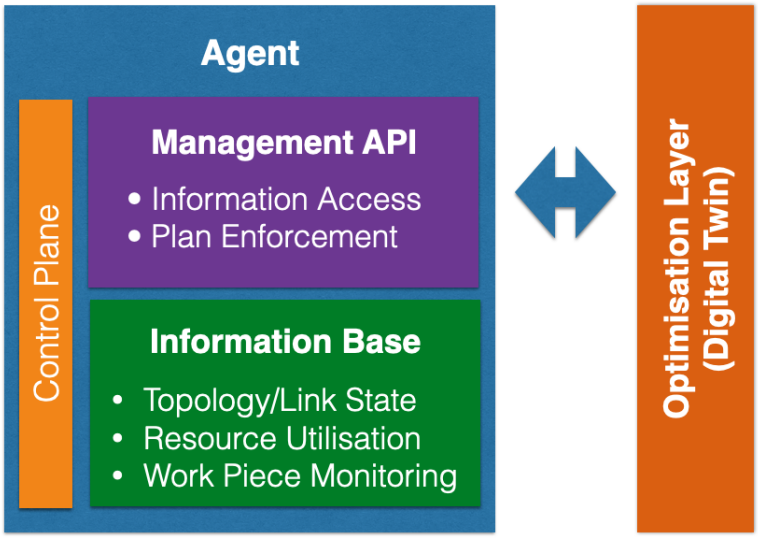


Figure 8: Piccolo Agent, Information Base and Optimization Layer

- Enforce workpiece flow as requested by optimization layers and digital twin

It is still not clear how much the optimization of workpiece flow will be part of the Piccolo Agent versus the inter-agent communication. It might be desirable to disconnect function placement and workpiece flow entirely since workpieces are part of a higher abstraction layer.

As explained in D2.1 [1] the Piccolo node will run on an industrial version of *GRiSP2* hardware platform (see Figure 9). The software platform is a unikernel consisting of the Erlang BEAM D2.1 [1, Section 2.6], OTP D2.1 [1, Section 3.3] and RTEMS [12].

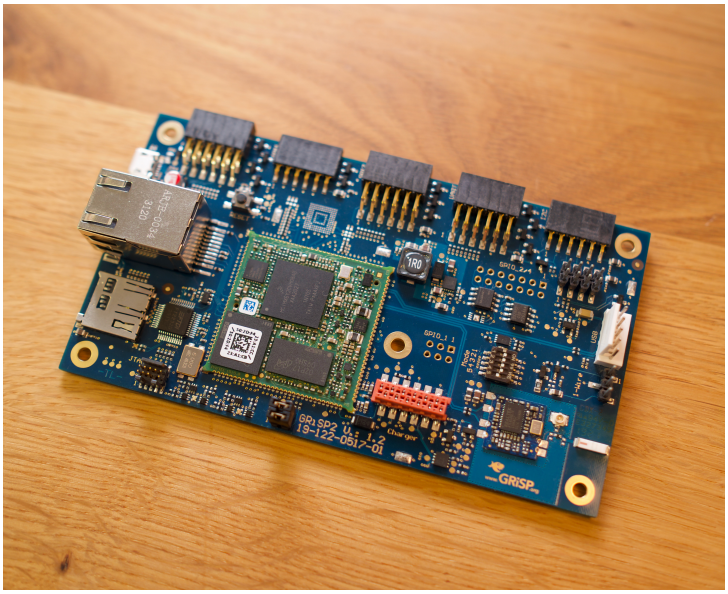


Figure 9: *GRiSP2* standard board (industrial version not manufactured yet)

Compared to node architectures described in D2.1 [1, Section 5.1] the Smart Factory use case will probably be the simplest manifestation. There is just one execution environment instance for Erlang

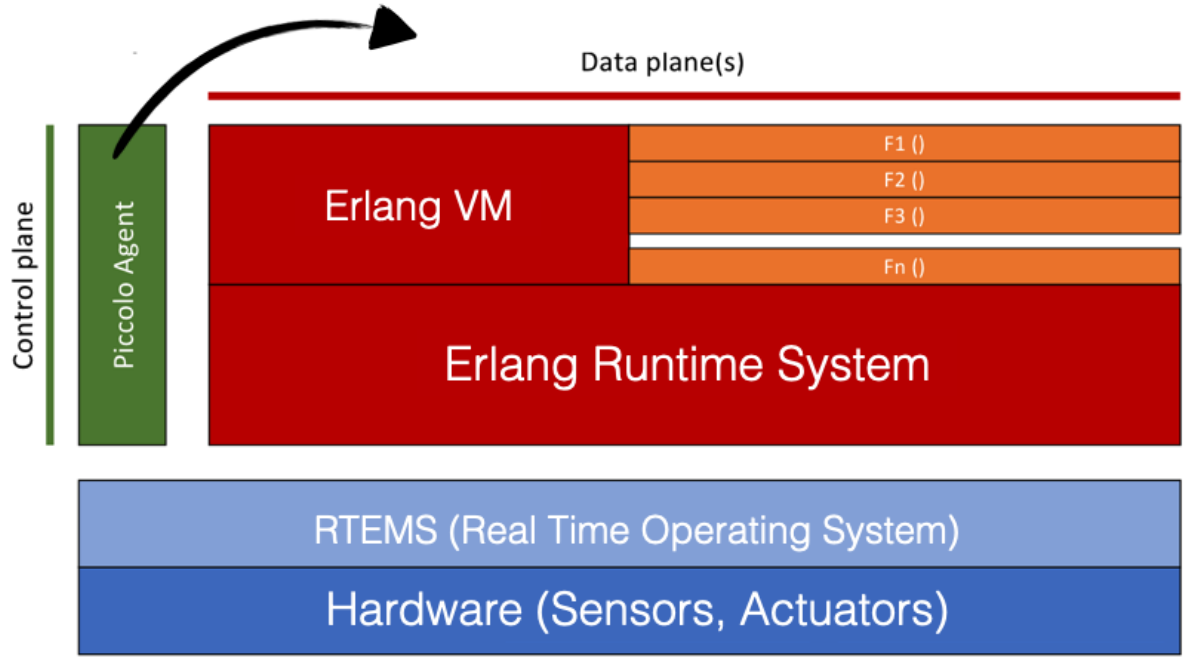


Figure 10: Smart Factory Node Architecture, Agent is also processed as Actor

actors on a single host, and within this architecture the Piccolo Agent is treated as any other Piccolo function – also as an actor. The use case is dealing with a closed system, hence software security currently can be neglected.

Since function placement is such a crucial part of the use case the question of distribution of Erlang BEAM executable byte code arises. OTP provides a server entity [13] which can be used to store and extract executable byte code. Distribution of byte code can simply be done by targeted Remote Procedure Calls (RPCs) or in a broadcast fashion to update all nodes in a topology at the same time. For a start it appears simpler keep all byte code updated on every node, e.g. every node has the same set of byte code fragments all the time. The byte code consists of Erlang modules representing function blocks of the topology given through IEC 61499. A Piccolo Agent and optimization layers can reconstruct that topology through meta information in every module to ensure communication between function blocks.

3.2.2 API considerations

The potential usage of APIs as described in D2.1 [1, Section 5.3] does not take a clear shape at the moment. Function placement will likely be optimized according to placement of sensors and actors. Hence (referring to Figure 3 API11 (node management interface to special hardware) will play a dedicated role while API10 (interface to secure hardware) will be completely neglected. Since there is just one execution environment instance API 3 and 4 (EE management) are irrelevant. All other APIs are important in the scope of function migration and discoverability within a network topology, and also establishing connections between functions as described through IEC 61499.

Several existing components of Erlang/OTP can be interpreted as implementation of the abstract APIs described in D2.1 [1, Section 5.3]:

- API1 (Agent-Runtime) and API2 (Agent-Execution environment) map to Erlang/OTP supervision tree, process registry and high level abstractions for server and event handlers – all of which are just actors reacting to messages. Note that Erlang applications are always organized in a process supervision tree where parent nodes are responsible for the lifecycle of their children.
- API5 (Runtime-Network) and API6 (Function-Network) map to the Erlang/OTP process registry and its native TCP based framework for discovering and accessing other Erlang nodes via messages. The Erlang code server is responsible for distributing byte code.
- API7 (EE/Runtime-function) maps to the lifecycle management capabilities of Erlang/OTP supervisor trees which include means to start and stop functions as well as failover strategies. It is also possible to hibernate functions to reduce memory footprints.
- API10 (Agent/node management-secure hardware) maps to the GRiSP2 API for its secure element, a hardware device that stores secrets and certificates, and is able to perform cryptographic operations on them. It is however not clear yet if and where it will be used in the use case.

All the above Erlang/OTP components are by default included into every Erlang application and they provide plugin mechanisms (in an inversion-of-control sense) to make use of them. In the upcoming process of developing the Piccolo node it will likely happen that abstractions will be found which are closer to the Piccolo API descriptions, moving away from the standard Erlang/OTP APIs but under the hood still using them.

IEC61499 function blocks and their state machines are compiled into Erlang modules as illustrated by the code snippet in Appendix A.2. Those modules contain callbacks for state changes which are 'hooked' into an Erlang actor on startup which again represents a small runtime for this particular function block. Since state changes can be manifold the callbacks can be very different between compiled function blocks. However, the API for starting the actual Erlang actor is very simple: there's a *start* function which takes the compiled IEC61499 Erlang module as argument and a corresponding *stop* function.

3.3 Piccolo docker node with hardware acceleration features

This section outlines the capability of a Piccolo node that has specialised hardware for dedicated tasks beyond data- or packet processing, specifically Time-Sensitive Networking. The corresponding PoC highlights the integration of a TSN sub-network with Industry 4.0 (I4.0) specific communication into the Piccolo ecosystem, built on hardware accelerated platforms, namely InnoRoute's research platform TrustNode and RealTimePI, shown in Figure 11 and Figure 12.

Figure 11: TrustNode TSN router as enabler device for the TSN OPCUA PoC

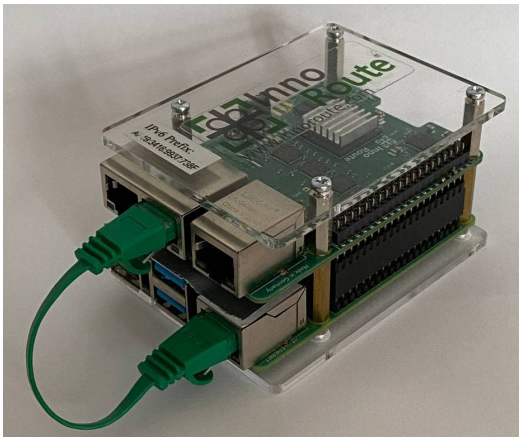


Figure 12: RealTimePI TSN endpoint as enabler device for the TSN OPCUA PoC

3.3.1 Basic Docker node

This describes the implementation of a basic docker engine software package on a node which exposes the real-time networking capabilities of the underlying hardware. Figure 13 shows a block diagram of a Piccolo docker node with access to specialised hardware. According to the definitions from D1.2 [1] the Piccolo agent becomes the control instance of the TSN node and the special hardware is the Time Aware Shaper (TAS), the entity which generates the reserved time slots for packets.

The Execution Environment entity is here specified as a docker engine. The Piccolo agent requests the API to run new instances of Piccolo applications on top of this EE. The runtime component is provided by the *docker-compose* toolchain. In this PoC the Piccolo agent also provides information about the specialised available resources of the Piccolo TSN node, which in this case are the TSN capabilities of the TrustNode or RealTimePI.

3.3.1.1 Piccolo agent on docker node

The Piccolo agent implementation enables it to deploy a new docker environment on the node. The node design strives for simplicity with enough flexibility to target further use-cases and extensions. This ensures the implemented API could merged with other Piccolo node API implementations later. The agent API is extended according to D1.2 [1] with functions to mange the Piccolo applications running inside this new docker environment. This includes functions to *CREATE, DELETE and LIST* new/running environments. If a new Piccolo application is instantiated, the required resources need to

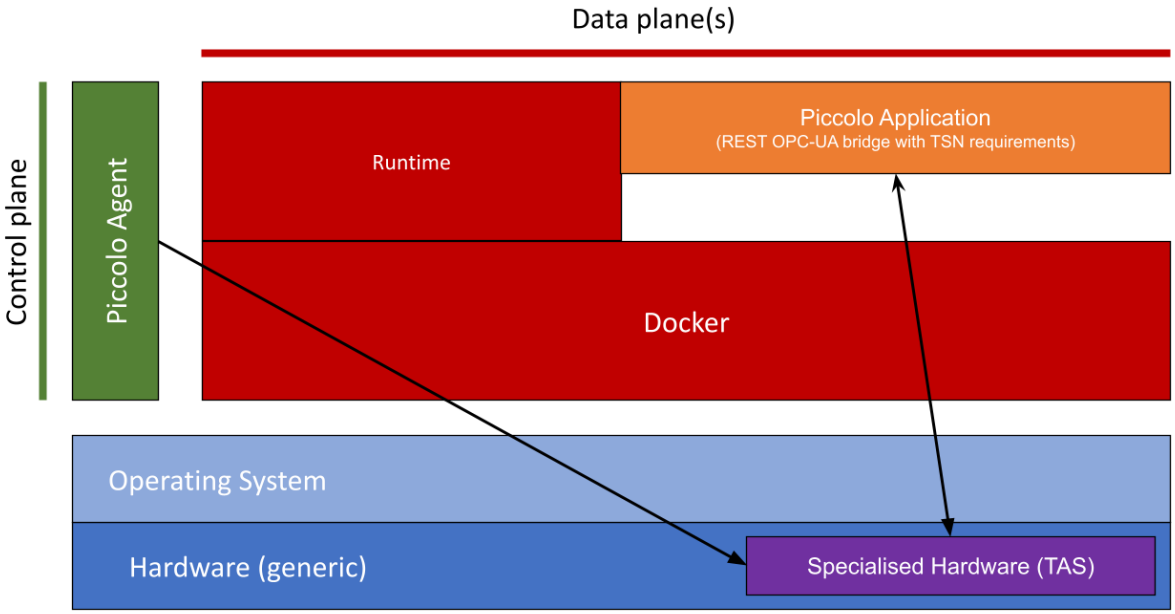


Figure 13: Basic docker node block diagram (with PoC specific instances loaded)

match the provided resources of the selected EE. Listing 2 shows an example instantiating request for this dedicated node type. The provided metrics ...*"type":"TAS","devicetype":"trustnode"*,... match to the provided capabilities of the Piccolo docker node running on the TrustNode device. The remaining definitions in Listing 2 will ensure the instantiation of the PoC-specific *innoroute/rest2OPCUA* container with hardware access to the Memory Mapped Interface (MMI) of the TrustNode to control the settings of the Time Aware Shaper. Additionally, the northbound interface type *rest2OPCUA* is advertised by the Piccolo agent. This enables external calls of the new function provided by this Piccolo node. The Piccolo API needs to provide information about running instances of Piccolo functions on the environment.

```
{"piccolo-object": {
"type":"docker",
"resources_required":{
  "accelerator":{"name":"TrustNode_TAS",
  "type":"TAS","devicetype":"trustnode",
  "interface":"mmi"}},
"name":"",
"interface":[{"type":"rest","subtype":"rest2OPCUA",
"TSN":{"max_latency_ns":5000, "bandwidth_kbs":20}
          }],
"config":"version: '2'\nservices:\n  test:\n\
    image: innoroute/rest2opcua:latest\n\
        ports:\n      - '56:5000'\n"
}}
```

Listing 2: Example of Piccolo Docker TSN node instantiation

In Appendix A.1, Listing 4 shows the state of the *docker_env_1* docker environment for a Piccolo node that is running one container named *Piccolo2OPCUA_1*. Due to the special requirements of the Piccolo function (*TrustNode_TAS*) the resources need also be present in the root of the Piccolo agent, otherwise the Piccolo function could not have been accepted for instantiation. The Piccolo API also provides information about the northbound interface of the running function, in this case this is an *rest2OPCUA* interface. The API provides the information required to communicate with this interface type. The agent itself is not involved in the data-plane communication. If another Piccolo function is able to deal with the interface type *rest2OPCUA*, just the network port parameter is needed to open a connection.

The specifics of the OPC Unified Architecture (OPCUA) interface are not part of the Piccolo environment. Additionally, the interface definition contains the *TSN* definition which is used to configure the special timing requirements of a connection.[10] For the docker environment, the Piccolo agent needs to manage all requests of external network ports for new instantiated Piccolo functions. As shown in Listing 2 the requested external port for the interface was *56*, the Piccolo agent has mapped this port automatically to the next free port in its available port range (*5001*).

3.3.1.2 Implementation details

Figure 14 shows the implementation details of the Piccolo docker node. The basic virtualisation

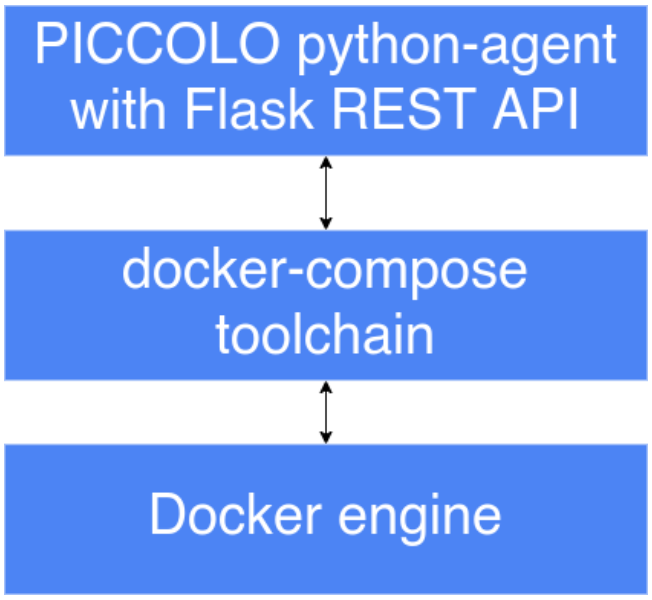


Figure 14: Block diagram of basic docker node's components

functions are provided by the docker engine itself. The toolchain of *docker-compose* is used to encapsulate the comprehensive docker functions into compact description files. The Piccolo agent is implemented in Python, which enables an easy adaption and extension with further features. The Python module includes the node's REST API, which was designed with support of *OpenAPI* tools.

[10] see D1.2 [3] for details.

With the described generic interface the Piccolo docker node environment is able to run generic Piccolo functions and manage their dedicated northbound interfaces to enable inter application communication.

3.4 μActor

μActor [14] is a serverless compute platform developed at Technical University of Munich that is available as open-source software[11]. Instead of the stateless functions used by state-of-the-art serverless systems, it uses actors as its unit of computation [14], which, as described in D3.1 [15], maintain local state and exclusively communicate using messaging. The platform aims to extend the concept of serverless computing from the cloud/edge-centric state-of-the-art towards the far edge and targets long-lasting applications spanning a heterogeneous set of devices [14].

The platform is generic, adheres to many of the concepts discussed in D2.1 [1], and could be used to implement many of the proof-of-concept applications discussed in D1.1 [2]. μActor's use has been demonstrated in a basic smart office scenario [14].

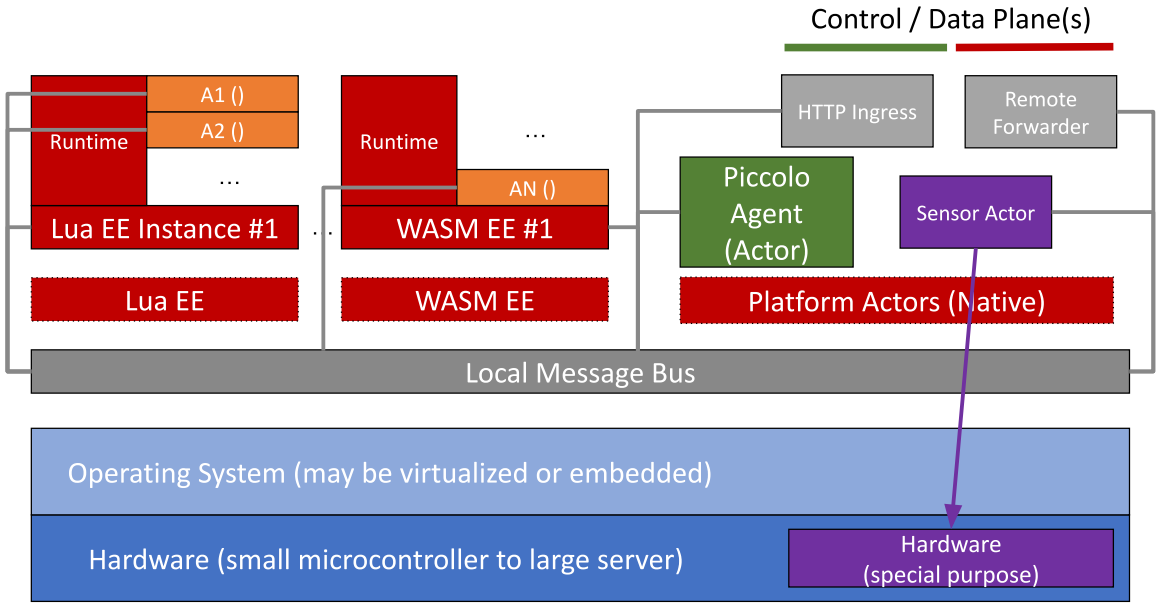


Figure 15: μActor as a Piccolo Node. Adapted from D2.1 [1].

μActor is a serverless platform and uses actors specified in a high-level language or a portable bytecode format, which are dynamically received from the network and executed using a language runtime that is provided by the system [14]. Similar to Cloudflare workers [16], μActor achieves the isolation required for *multi-tenancy* operation by sandboxing the actors using one or more language virtual machines [14], which correspond to the execution environments presented in D2.1 [1]. Figure 15 shows

[11] https://github.com/uActor/uActor

a high-level overview of a μActor node that was derived from Figure 11 in D2.1 [1]. The execution environments and their corresponding runtimes are shown as the red boxes in the top-left of the figure and the serverless actors are shown as orange boxes.

The initial prototype uses a single execution environment for actors defined using Lua [14]. As shown in the figure, one could add additional execution environments, e.g., for actors defined using WebAssembly (WASM). It would be possible to dynamically create additional instances of a type of execution environment, e.g., the Lua execution environment, and implement an API corresponding to the **Execution environment to Execution Environment instance (plus runtime)** API. Furthermore, one could extend the system to manage the resource distribution between multiple execution environments and implement the **Agent-Node management** API.

The execution of the actors, which correspond to the Piccolo functions, is only driven by incoming messages, which allows the runtime to exercise precise control over their execution and multiplex many actors onto a single system thread [14]. Furthermore, the dynamically received actors only have a very limited set of functionality [14]. While the initial prototype requires the actors to cooperatively exit after processing each message with a limited set of instructions, the publication [14] discusses various ways to allow for blocking and long-running actors. This is required to achieve the *security* goals or the preemption of a function to be able to run a higher-priority function that are discussed in D2.1 [1]. This approach to scheduling can be interpreted as an implementation of the **EE/Runtime-function** API.

The actors communicate using content-based networking [17, 18], which routes messages based on their content. Similar to existing systems [18], μActor uses messages consisting of typed key-value pairs and subscriptions consisting of constraints on these fields [14]. This allows for simplified extension — extensions can add subscriptions without requiring a change to the publisher — and decoupled (*location-independent*) communication [14]. The actors (runtimes) publish a set of (default) subscriptions, which are used across the network to route messages towards interested actors [14]. This can be interpreted as an implementation of the **Function-Network** and **Runtime-Network** APIs. μActor uses a node-local message bus that routes the message between local actors and the remote forwarders, which are platform actors (defined below) that can be implemented using *multiple underlying communication protocols* and forward the messages between multiple nodes [14]. Those platform actors implement a dissemination strategy deciding which messages and subscriptions are forwarded across node boundaries [14]. The networking components are shown in grey in Figure 15. Messaging is the only way for actors to interact, both on a local node as well as across multiple nodes. Therefore, the content-based messaging covers both the **Runtime-function-function** API as well as the **Function Protocols**.

The same actors can be deployed to a variety of devices, both powerful Linux nodes supported by other serverless platforms as well as small, resource-constrained microcontrollers such as the ESP32 [14]. In the context of Piccolo, the μActor runtime could also be nested inside an execution environment for Containers or Virtual Machines (VMs).

Special node and hardware functionality is integrated into the system using *platform actors* that are

implemented as part of runtime but interact via messaging [14]. Therefore, the system resources are available as actors in a platform-independent way. As shown on the right side of Figure 15, these Platform actors can be used to integrate, e.g., sensors, external communication protocols such as HTTP, or any other functionality beyond basic, isolated computation and messaging [14]. As they are statically included as part of the platform and have exclusive access to the system resources, there is no requirement for an **Agent/Node Management - special Hardware** API, and the **Agent-Node management** API would be limited to managing general-purpose hardware. In summary, the platform is highly suitable for *multi-platform operation*.

In addition to the fine-grained scheduling discussed above, the serverless approach also provides the node with many capabilities related to the *automatic management of resources and failures* discussed in D2.1 [1]. As the actors are only active while processing a message, only have a very narrow interface, and each actor's state is transparent to the execution environment, the execution environment gains flexibility in managing the actor's lifetime and storage [14]. This allows controlling their resource footprint using hibernation and allows for the migration of actors [14]. Furthermore, this provides the execution environment with precise information of an actor's resource consumption and allows to assess the *capacity* of a node in a fine-grained way. As the platform actors only react to messages, they can ensure safe access to special hardware resources.

Actors are able to spawn and manage other actors, which allows them to be used to implement an orchestration system [14]. The agent functionality can, therefore, be implemented as a set of actors interacting using (content-based) messaging. Figure 15 shows the Piccolo agent as a platform actor — large parts of the agent functionality may, however, also be implemented as an actor executed using one of the execution environments. Most of the management APIs defined in D2.1 [1] can therefore be implemented by letting the system components, e.g., the actors and the execution environment, emit and listen to events. Hence, e.g., the **Agent-Runtime**, **Agent-Execution environment**, **Agent-environment**, and **Execution environment to Execution Environment instance (plus runtime)** APIs as well as the **Agent Protocol** and **Runtime Protocols** can be implemented using actors and messaging. The agents on multiple nodes can communicate with their peers on other nodes using the same messaging mechanism. Hence, there is no need for a separate *Control Plane*.

As an example, the initial prototype contains a deployment system that is implemented using actors and allows to deploy actors to nodes as soft-state [19]. Listing 3 shows a JSON representation of an example deployment message (reduced to the fields relevant to the deployment system) published to the network and received by the *deployment manager* on the nodes. The *deployment manager* is implemented as an actor. The deployment is specified using a *type* field, a *name*, and a *lifetime* (lines 2–4). Furthermore, the message contains information such as the *name*, *version*, and the *type of required execution environment* and (optionally) the *code* of the actor (lines 6–9). Finally, the set of nodes the deployment (actor) should run on can be limited by requiring a *set of actors* running on the node and a *set of labels* attached to the node (lines 11–13). The latter are included as optional constraints in the deployment subscription and are therefore used to limit the spread of the deployments throughout the network. Internally, the management actor maintains information about the deployments, starts and stops the actors by sending messages to the execution environments, and receives updates about their status as messages published by the actors and execution environments.

```
{
    'type': 'deployment',
    'deployment_name': 'de.tum.ping_pong.deployment.ping',
    'deployment_ttl': 60000,

    'deployment_actor_type': 'de.tum.ping_pong.actor.ping',
    'deployment_actor_runtime_type': 'lua',
    'deployment_actor_version': '1.0',
    'deployment_actor_code': 'function receive(publication)...',

    'deployment_required_actors': 'com.example.actor',
    'deployment_constraints': 'foo',
    'foo': 'bar',

    // ...
}
```

Listing 3: Example μActor deployment message

This orchestration system is further discussed in [19].

Generic protocols that support the integration of multiple Piccolo node-types could be implemented using platform actors, e.g., one that converts HTTP requests into messages that are then processed by an actor as it is shown in Figure 15. Similarly, external runtimes such as a Docker daemon could be exposed to the actors using platform actors and, therefore, enable the management of other types of Piccolo functions using dynamically deployed actors.

4 PoC Isolation and Security discussion

Providing robust isolation between applications running on the same Piccolo node is a key requirement previously described in D2.1 [1]. This is essential from a security point-of-view where the prospect of accidental or deliberate leaking of information from one application to another could render the platform untenable. Applications from different users (or the same user) also compete for the resources of the Piccolo node (for example: CPU cycles, memory, network) and the allocation of these resources must also be robustly enforced.

4.1 Lightweight Virtualization

We are exploring the use of using Kata Containers, a lightweight virtualisation technology which provides a stronger degree of isolation for Linux containers. Kata Containers use hardware-virtualization to run a dedicated kernel for each container or pod (a set of containers) providing isolation of network, I/O and memory and can utilize hardware-enforced isolation. The kernel that is executed for each container is not the same as the host-kernel and a simpler root file system is provided [20]. Figure 16 shows how traditional containers compare with Kata Containers [21].

Kata Containers supports a number of different hypervisor backends including QEMU/KVM, Amazon Web Services (AWS) Firecracker [22] and cloud-hypervisor [23].

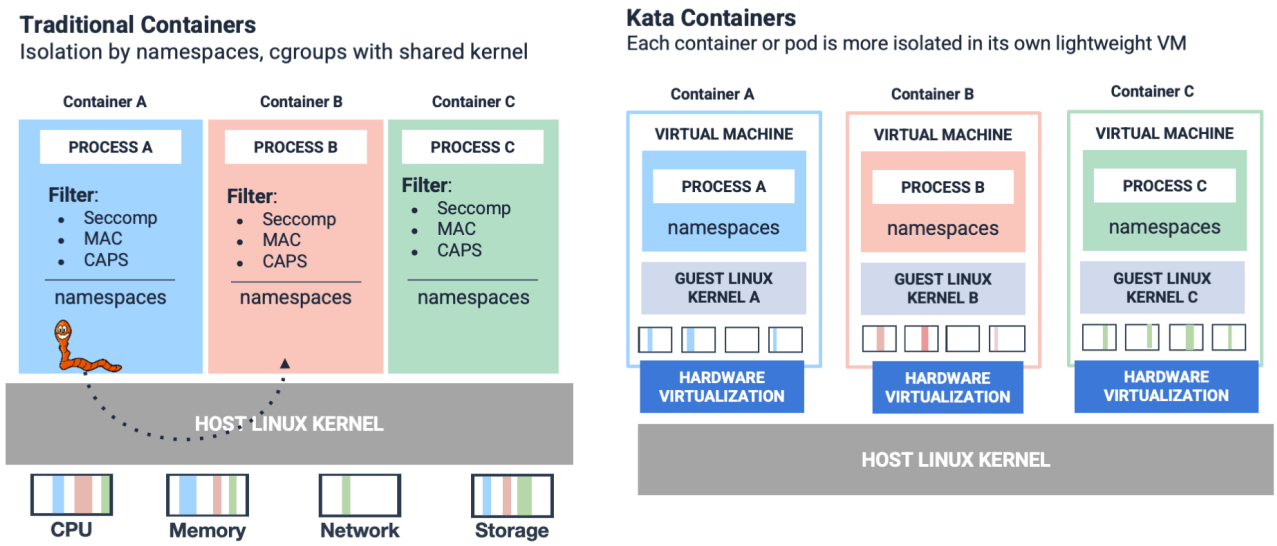


Figure 16: Comparison of native containers vs Kata Containers

Using Kata Containers extends the isolation provided in the case where a Piccolo application is deployed as a container using a container runtime, such as docker or containerd, as the EE. Each Piccolo application deployed using Kata Containers is isolated within a separate Lightweight Virtual Machine (LWVM) as shown in Figure 17.

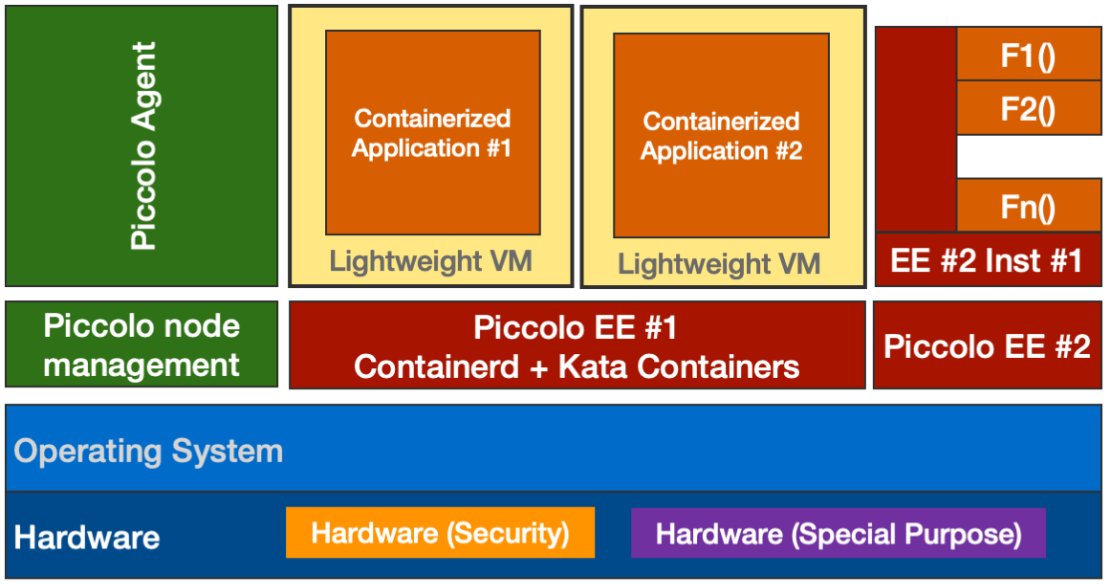


Figure 17: Using Kata Containers as an Execution Environment

Figure 18 shows an alternative scenario that isolates each instance of an EE on the Piccolo node by deploying each EE within a container inside a separate LWVM. This makes it possible to use multiple instances of an EE that does not natively support multi-tenancy.

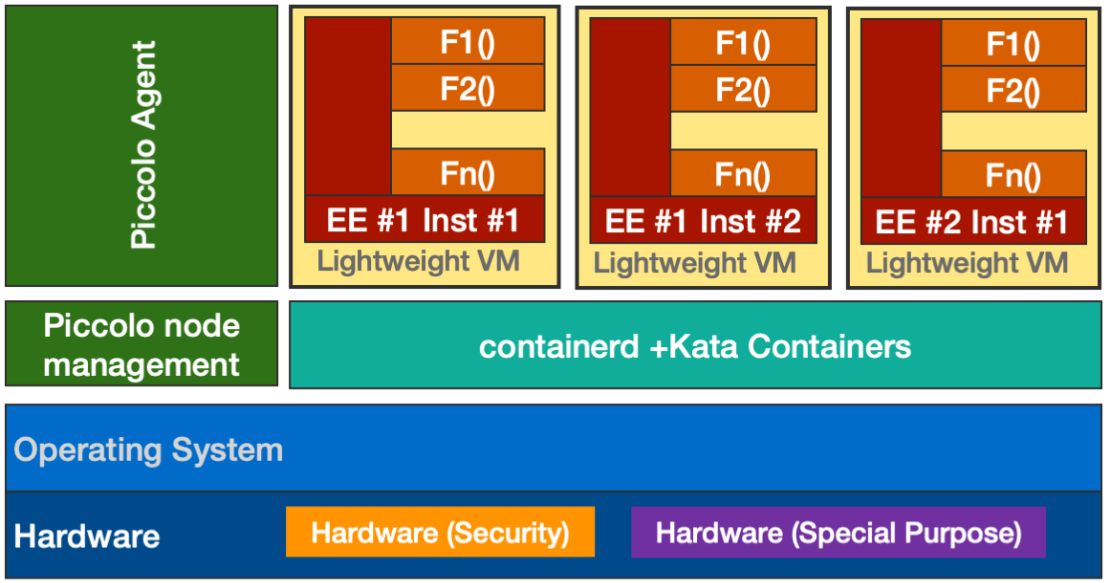


Figure 18: Using Kata Containers as system component to deploy Execution Environments

4.1.1 Results

Our initial experiments used Kata Containers v1.12 with docker on a number of different Arm-based platforms and we obtained preliminary results comparing network latency and bandwidth for contain-

ers deployed using runc and Kata Containers using QEMU/KVM. The configuration of Kata Containers was not optimized (we used the out-of-the-box defaults as far as possible). The early results across the different hardware platforms showed a network latency increase between 36 % and 147 % and an up to 63 % reduction in maximum sustained bandwidth when using Kata Containers.

4.1.2 Further work

Kata Containers v1 is now deprecated and Kata Containers v2 is now the supported version. Kata Containers v2 includes changes that have made it necessary to alter aspects of our experimental setup. Kata Containers v2 works only with container runtimes that use v2 of the runtime shim API and therefore does not work with docker (which only supports v1 of the runtime shim API). Kata Containers v2 includes *containerd-shim-kata-v2* which enables us to use containerd in-lieu of docker.

We intend to repeat the network latency and bandwidth experiments using Kata Containers v2 along with measuring other properties such as memory usage and filesystem performance. We will also use firecracker as an alternative hypervisor for Kata Containers. We will include this new configuration in our benchmarking.

4.2 Behavioural Risk Monitoring Proof of Concept - Security Concepts

The TINC/Piccolo Node provides operations and technologies to protect data and code at rest, in transit and in use. While shielding data at rest and in transit involves the adoption of encryption mechanisms in order to achieve confidentiality and integrity of data, securing data in use (i.e. when code and data are loaded in memory unencrypted) is even more challenging. The TINC/Piccolo node leverages from Trusted Execution Environments (TEEs) [4] and in particular from the Intel SGX [5] technology combined with the SCONE framework [6]. This framework offers a set of utilities and operations that not only simplify the interactions with the hardware (Intel SGX driver) but also generate confidential applications without complex configurations. In addition, to protect against vulnerabilities such as deprecated firmware, software or hardware components the system leverages from an important aspect of confidential computing, called attestation. This operation ensures the confidentiality and integrity of the function. Components like the CPU, its firmware and the function's code and data itself are being attested prior to execution. SCONE provides a seamless attestation mechanism that is integrated to the TINC/Piccolo node in order to be transparently performed at each function startup.

The Behavioural Risk Monitoring PoC involves pipelines with multiple stakeholders such as the service provider (Bosch or Sensing Feeling), the infrastructure provider (Fluentic), and the end-users (e.g. car drivers). From the end-user's viewpoint, securing any private data from leaking to unauthorized entities is essential. On the other hand, the service providers require that their business logic remains protected (i.e. no other entity can manipulate their data and code, even the infrastructure provider) and act as trustworthily as possible towards their users. Therefore, it is crucial to utilize

protection mechanisms that isolate critical procedures during computation and ensure data and application code safety for multiple stakeholders. Although we have already described in Section 3.1.2 the operation called *Tincify* that converts a native compressed function into a trusted containerized function, in this section we focus more on the aspects of attestation, the generation of session policies, management of function secrets, such as encryption keys, certificates and configuration parameters, and the actual execution of the function in a TEE.

Attestation and Configuration Procedures

There are cases where enclaves need to collaborate with other enclaves on the same platform due to data exchange in case of limited enclave space to retain all the information or with Intel's reserved enclaves to request Intel related operations. Therefore, it is crucial to accomplish trust between the two enclaves. On the other hand, there are times that a client needs to prove (mainly to a server of the application provider or vendor) that the function is running on a trusted platform that can operate on secrets securely. Both the aforementioned conditions require a proof of a TEE and the process of claiming that is defined as attestation, which is divided into two types:

- **Local Attestation**: a successful result of a local attestation yields an authenticated affirmation between two enclaves running on the same platform that trust has been established and that they can transfer exchange data securely.

- **Remote Attestation**: the same operation as the local attestation but now a client needs to provide this kind of verification to a server so that the latter can provide the former with the secrets it requested in an assured manner. In more detail, a remote party compares the enclave measurement[12] [24] reported by the trusted platform with an expected measurement, and only proceeds if the two values are equal.

The TINC/Piccolo node leverages from two services that are executed inside enclaves and implement the local and remote attestation procedures without the need for source code changes on functions. The pipeline of the PoC assumes that the remote attestation service[13] is trusted [14] and can be located at the infrastructure provider's site or even the application provider's domain. The local attestation service resides inside an enclave and performs the operations needed for the function's enclave to accomplish attestation and secrets acquisition upon bootstrapping.

Session Policies: On its own, the attestation procedure does not offer flexibility and control over operations that a function may require like acquisition of encryption keys. Therefore, a TINC/Piccolo node utilizes a scheme of a session policies provided by SCONE. A session policy is a subset of YAML[15] and is defined in a TINC/Piccolo node for each function to cater for each function's secrets

[12] An enclave measurement can contain code, data, heap, stack, location of each page within the enclave and security flags employed.

[13] There can be multiple attestation services but then it needs to be defined which one should be used each time. Therefore this expansion is not covered in the current PoC.

[14] If the remote attestation service resides on an untrusted server, then the application owner should perform a remote attestation of that service before transmitting encryption keys and certificates to it.

[15] https://yaml.org/

and measurement values. Additionally, this policy can define which functions are allowed to communicate by providing their session identifiers. The session policy is taken care by the TINCmate utility and abstracts all the operations offered by SCONE's utilities.

Function Secrets: The session policy can also maintain a function's secrets (like encryption keys) that are transferred through Transport Level Security (TLS) towards the Remote Attestation Service. Secrets are uniquely defined by a name (in a key-value manner) and can be either provided to the policy by the function developer or generated by the attestation service, which reduces risks of secrets being revealed to humans. Those secrets are only allowed to be loaded to a function upon request and after it has a valid function measurement value that identifies that the function is able to run on the designated trusted platform.

Secure Function Execution: The execution of the function (e.g. *IoTea* Talent) is done through the bootstrapping procedure of a docker container. The container will initiate first the "Observer" module and then the actual function. Both of these modules run inside enclaves and in order to be executed they need to be attested by a remote party and upon success receive all the secrets needed. Otherwise any encrypted information cannot be decrypted inside the secure execution environment. Moreover, the two modules, "Observer" and the function, can communicate with each other without the "Observer" module being able to intercept sensitive data destined for the actual function [16].

The above features are depicted in Figure 19 that presents an example of converting a native application into a TINC/Piccolo function along with attestation procedures. The operation starts with the native application and its data/libraries needed and the "Tincify" step that takes place. This step generates an encrypted docker image and produces a session to a remote third service (challenger). The remote service runs also inside an enclave (so it can also be attested as well) and can store secrets of the functions. The last step is to deploy this produced encrypted docker image to a Piccolo Node where the validation procedure takes place in order to receive secrets and execute the function.

4.2.1 Next Steps

The support of attestation and isolation through Intel SGX and SCONE for now is implemented using Docker containers and a known (by all stakeholders) trusted attestation service. One challenge worth exploring is whether such integration can take place using Kata Containers where the isolation mechanisms are more fine-grained and provide greater level of security as described above.

Furthermore, certain nodes (Intel NUCs) on the PoC are configured to use Intel SGX v1, which requires to define details of the enclave such as heap and stack size prior to its execution. This incurs a burden for the application/service provider who needs to attempt different values before reaching an appropriate one for its trusted computing. Our next step is to configure the nodes to support Intel SGX v2 which provides a dynamic runtime configuration without the need to state the aforementioned properties of an enclave.

[16] The configuration for that policy is specified in the policy that is generated during the *Tincify* process.

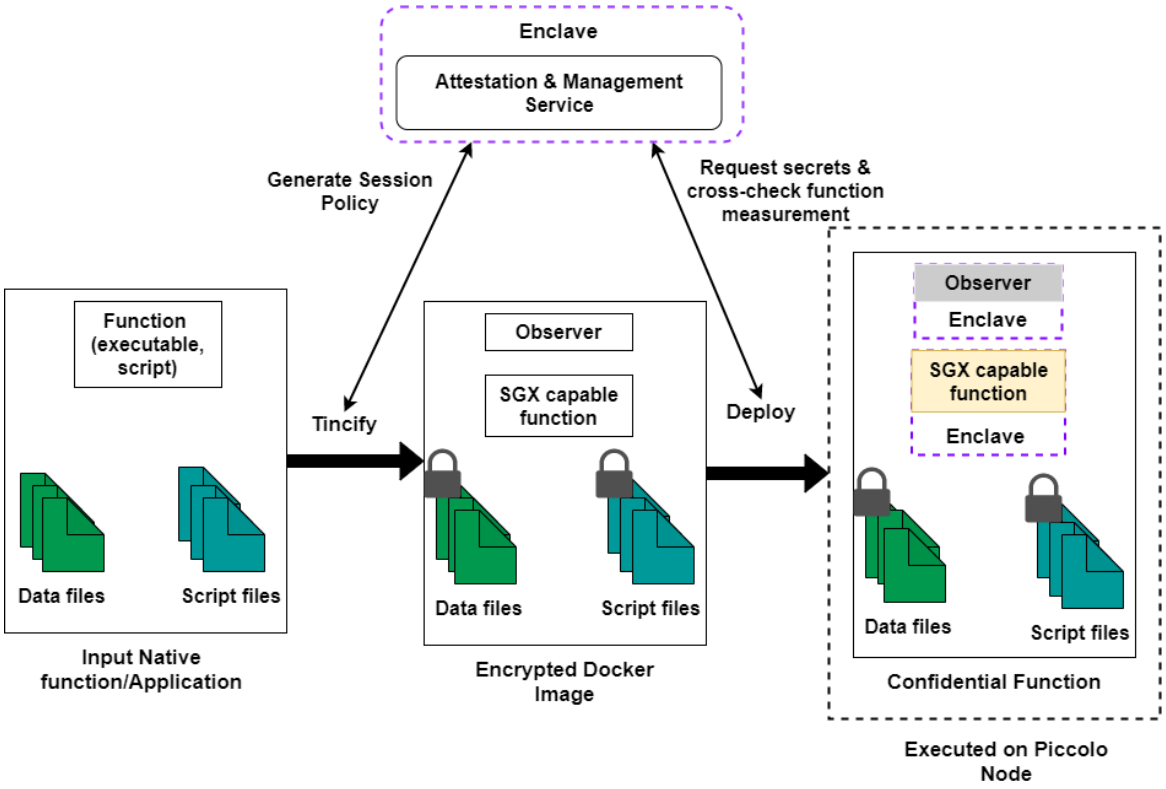


Figure 19: Converting a native application to a TINC/Piccolo function along with attestation operation.

5 Conclusion

In this report we have shown how each of the Proof of Concepts map the ideas described in the initial node architecture to real-world scenarios. No single Proof of Concept covers every aspect of the architecture but as the implementations mature, via the next steps described below, we expect a more complete picture of the requirements and constraints to emerge. This will enable us to refine elements within the architecture specification, such as the various APIs, whilst retaining the flexibility to address a wide-range of potential use-cases. This report also shows that the implementation of a Piccolo node is not limited to a specific class of devices - nodes may be small embedded-systems or even server-class machines running containers or entire virtual machines. There is a delicate balance to be achieved in defining how much detail is required in the Piccolo APIs to enable effective deployment of applications across this expanse of capabilities.

Next steps and challenges:

Behavioural risk node: Our primary next step is to investigate the usage of the Piccolo API from the evaluation of the first Proof of Concept deployment for both of the processing pipelines (i.e. the vision processing pipeline and the in-vehicular data processing pipeline). Furthermore, we aim to provide better support to application providers in function deployment and management while simultaneously allowing for more complex and generic types of workloads. Additionally, we plan to research and support schemes for improved failure management and checkpointing operations as well as enhance our network stack by leveraging from the Information-Centric Networking (ICN) domain.

Smart factory node: Efforts are currently focused on a simulator application instead of the actual node implementation. For the node it is possible to re-use the components of the simulator (which is already a distributed application consisting of nodes) and hence development will begin with a migration of these components into a dedicated standalone application. In particular large parts of the simulator can be converted into a digital twin component.

Time-Sensitive Networking & docker node: We will continue enhancing the basic docker node implementation on the TrustNode platform to implement Time-Sensitive Networking for an in-network computing Proof of Concept. The next steps will be to include all the elements in a setup, with verification checks of the Piccolo agent implementation and accelerated hardware components.

μActor: We plan to improve the initial system by adding executors for additional languages and bytecodes, e.g., WebAssembly, automating the management of the network topology, and implementing various security features. Furthermore, we plan to improve the developer experience by providing additional tools and documentation. Finally, we plan to experiment with various actor-based orchestration systems and improve the scalability of the approach in larger networks [14].

Isolation and security: Next steps include measuring the cost of using Lightweight Virtual Machines to provide increased isolation over native containers. Further ahead, we want to look at how features of the Arm Confidential Computing Architecture could be utilized to provided the increased isolation and security with lower overheads than is currently possible. There will be challenges to overcome as this work will have to be done using models because hardware is unlikely to be available in time.

References

[1] 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.

[2] 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.

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

[4] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted execution environment: What it is, and what it is not". In: *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015* 1 (Dec. 2015), pp. 57–64.

[5] *Intel® Software Guard Extensions*. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html (visited on 03/05/2021).

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

[7] *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 09/06/2021).

[8] *OpenVINO™ Toolkit Overview - OpenVINO™ Toolkit*. URL: https://docs.openvinotoolkit.org/latest/index.html (visited on 09/06/2021).

[9] Piccolo Project. *Piccolo Initial Infrastructure Architecture*. Deliverable D3.2. 2021.

[10] *Sconify - SCONE Confidential Computing*. URL: https://sconedocs.github.io/sconify/ (visited on 09/17/2021).

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

[12] *RTEMS*. URL: https://www.rtems.org/ (visited on 03/03/2021).

[13] *Erlang Code Server*. URL: https://erlang.org/doc/man/code.html (visited on 03/03/2021).

[14] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. "μActor: Stateful Serverless at the Edge". In: *1st Workshop on Serverless Mobile Networking for 6G Communications*. MobileServerless'21. ACM, 2021.

[15] 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.

[16] Cloudflare, Inc. *Cloudflare Workers*. 2021. URL: https://workers.cloudflare.com/ (visited on 09/06/2021).

[17] Antonio Carzaniga and Alexander L. Wolf. "Content-Based Networking: A New Communication Infrastructure". In: *IMWS 2001*. Springer, 2001.

[18] Antonio Carzaniga and Alexander L. Wolf. “Forwarding in a Content-based Network”. In: *SIGCOMM ’03*. ACM, 2003.

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

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

[21] *Traditional vs Kata*. URL: https://katacontainers.io/learn (visited on 09/21/2021).

[22] Amazon Web Services. *Firecracker*. 2021. URL: https://firecracker-microvm.github.io (visited on 08/31/2021).

[23] *What is Cloud Hypervisor?* URL: https://github.com/cloud-hypervisor/cloud-hypervisor#1-what-is-cloud-hypervisor (visited on 09/21/2021).

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

Annex

A Appendix

A.1 Example: configuration of Docker Time-Sensitive Networking node

```
[
  {
    "envname": "docker_env_1",
    "objects": [
      {
        "type":"docker",
        "resources_required":{
          "accelerator":{"name":"TrustNode_TAS",
          "type":"TAS","devicetype":"trustnode",
          "interface":"mmi"}},
        "name":"Piccolo2OPCUA_1",
        "interface":[{"type":"rest","subtype":"rest2OPCUA","port":5001,
        "TSN":{"max_latency_ns":5000, "bandwidth_kbs":20}}],
        "config":"version: '2'\nservices:\n  test:\n\
            image: innoroute/rest2opcua:latest\n\
            ports:\n      - '56:5000'\n"
    },
    ],
    "type": "docker",
    "version": "2"
  },
  {"resources_provided":{
    "accelerator":{"name":"TrustNode_TAS",
    "type":"TAS","devicetype":"trustnode",
    "interface":"mmi"}}
  }
]
```

Listing 4: Piccolo Docker TSN node running configuration example

A.2 Example: IEC 61499 function block

```
FUNCTION_BLOCK yellow_green

EVENT_INPUT
    set WITH yellow, green;
END_EVENT

EVENT_OUTPUT
```

```
    done WITH r, g;
END_EVENT

VAR_INPUT
    yellow : INT;
    green : INT;
END_VAR

VAR_OUTPUT
    r : INT;
    g : INT;
END_VAR

EC_STATES
    ini;
    start : map -> done;
END_STATES

EC_TRANSITIONS
    ini TO start := set;
    start TO start := set;
END_TRANSITIONS

ALGORITHM map in ST:
    g := green + yellow;
    r := yellow;
END_ALGORITHM

END_FUNCTION_BLOCK
```

Listing 5: Structured Text representation of an example function block

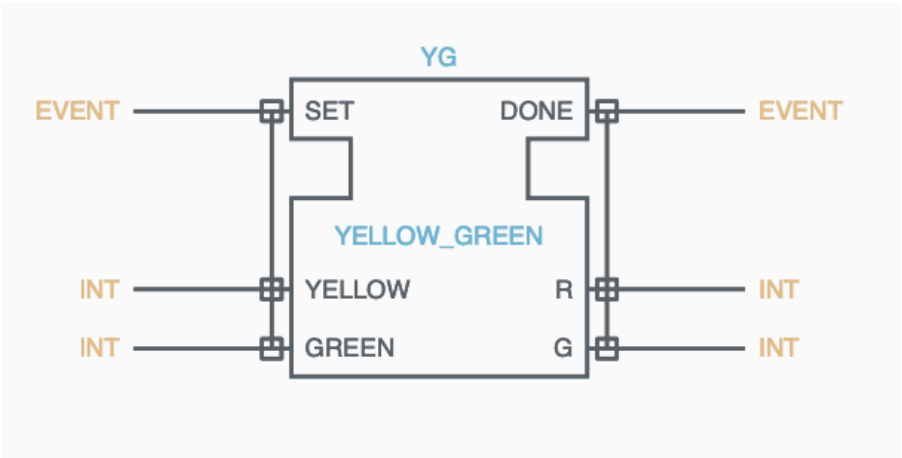


Figure 20: Graphic representation of the example function block

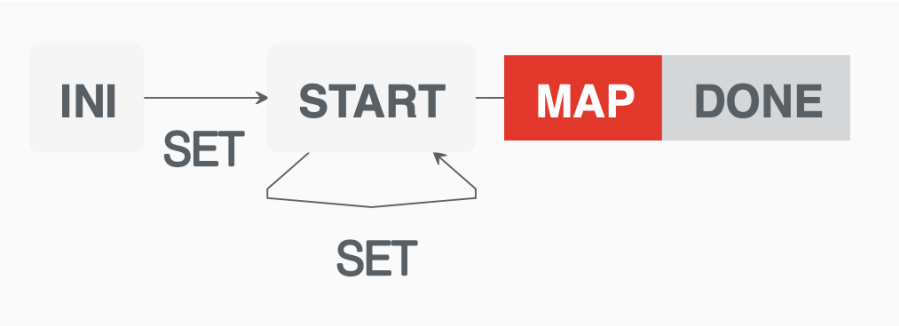


Figure 21: Grapic representation of the state machine of the example function block

```
1> yellow_green:module_info(functions).
[{module_info,0},
 {module_info,1},
 {ec_states,0},
 {ini,2},
 {start,2},
 {var,1},
 {event_in,0},
 {event_out,0},
 {event_out,2},
 {get_handler,0},
 {map,1},
 {fb_insts,0},
 {event_conns,0},
 {data_conns,0}]
```

Listing 6: List of Erlang functions in the compiled function block

This functions get called by our IEC 61499 runtime that starts a process for each function block.