D4.1.1 - Interfaces Definition

Delivrable number: D4.1.1
Work package: WP4
Version: 0.1
Deliverable Lead Org: Orange
Deliverable type: R
Dissemination Level: PU (Public)
Date: November 25, 2022

Authors:
Chrystel GABER (Orange), Nicolas Dejon (Orange), Samuel Legouix (Orange), Koen Zandberg (INRIA), Emmanuel Baccelli (INRIA), Damien Amara (Université de Lille)

Abstract:
The purpose of this document is to define the general architectural principles of a tinyPART platform. It describes the main components as well as their interfaces. The implemnentation and integration strategy of TinyPART to achieve the highest value of the project is described.

Keywords: Architecture, Interfaces, TinyPART

Acknowledgment The research leading to these results partly received funding from the MESRI-BMBF German-French cybersecurity program under grant agreements no ANR-20-CYAL-0005 and 16KIS1395K. The paper reflects only the authors' views. MESRI and BMBF are not responsible for any use that may be made of the information it contains.

Document Revision History

Document revision history			
Version	Date	Description of change	Contributor(s)
v0.1	15/10/2021	Creation	Chrystel Gaber
v0.2	04/12/2021	Update architecture and plan with partners comments	Chrystel Gaber
v0.3	12/01/2022	Fill Pip services and API sections	Nicolas Dejon
v0.4	22/04/2022	Update architecture and plan. Add introduction and section 1	Chrystel Gaber
v0.5	25/07/2022	Update sections 2.4 and 3.4	Chrystel Gaber

Table 1: Document revision history

Contents

Introduction

This deliverable D4.1.1 reports the work performed in the context of Task 4.1. This work performed at the beginning of TinyPART defines the architectural objective of the project and will serve as a guideline throughout it. This work brought together the contributors of Work Packages 1, 2 and 3 to aggregate their vision.

TinyPART's objective is to build technology bricks to help IoT developers conceive devices which are private-by-design to address at best the European Union's requirements on security and privacy while remaining competitive by minimizing their time to market and development costs.

The approach proposed by TinyPART consists in combining isolation of the addressing space, runtime isolation, confidentiality or integrity of requests to start/stop/activate the container as well as differential privacy. The most important requirements for this architecture are its modularity, its optimization of memory foorprint and energy consumption.

The context surrounding TinyPART is related to multi-tenant devices. As illustrated in figure 1, we suppose that IoT Device Owners allow third parties (Service Providers A, B and C) to deploy scripts on IoT devices in order to offer services to some end users through the use of a Script & Device Management Platform. They can also delegate the maintenance of their IoT device to a Maintainer through the same platform.

The project TinyPART investigates mechanisms to isolate within IoT devices untrusted application logic from multiple parties as well as on-board privacy pre-processing mechanisms. The Script & Device Management Platform is out of the scope of our works. Therefore, the descriptions provided in this deliverable focus on the mechanisms within the IoT devices. We will evoke functions of the Script & Device Management Platform, only if it is necessary to explain some proposed mechanisms within the IoT devices.

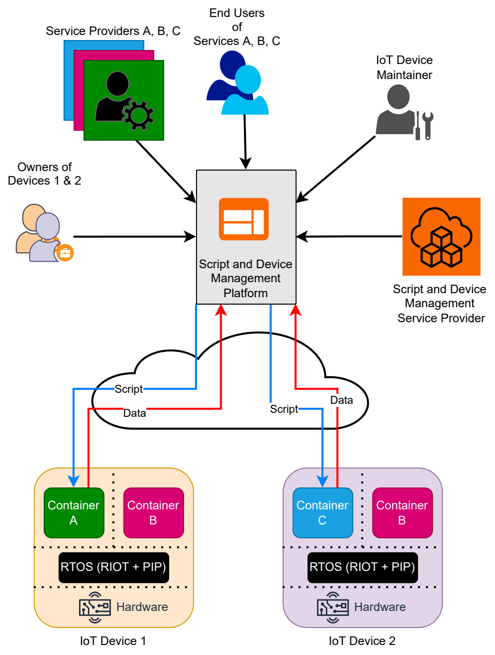

Figure 1: TinyPART usage context

The deliverable is organized as follows. Section 1 presents an overall view of the architecture as well as the iterative development and integration strategy. Sections 2 and 3 describe respectively the modules and their interfaces involved in the architecture.

1 Overview of the architecture

The overall architecture featuring all components is represented in figure 2. It is composed of four modules. First, the PIP protokernel adapted to use the Memory Protection Unit (MPU) provides strong isolation guarantees. Second, LIBPIP acts as a linchpin between PIP and RIOT. The third module, namely the Runtime Containers Service provides all the primitives for secure container management and secure actions triggering within the containers. Finally, the Differential Privacy Service provides mechanisms to anonymize data collection directly on the device.

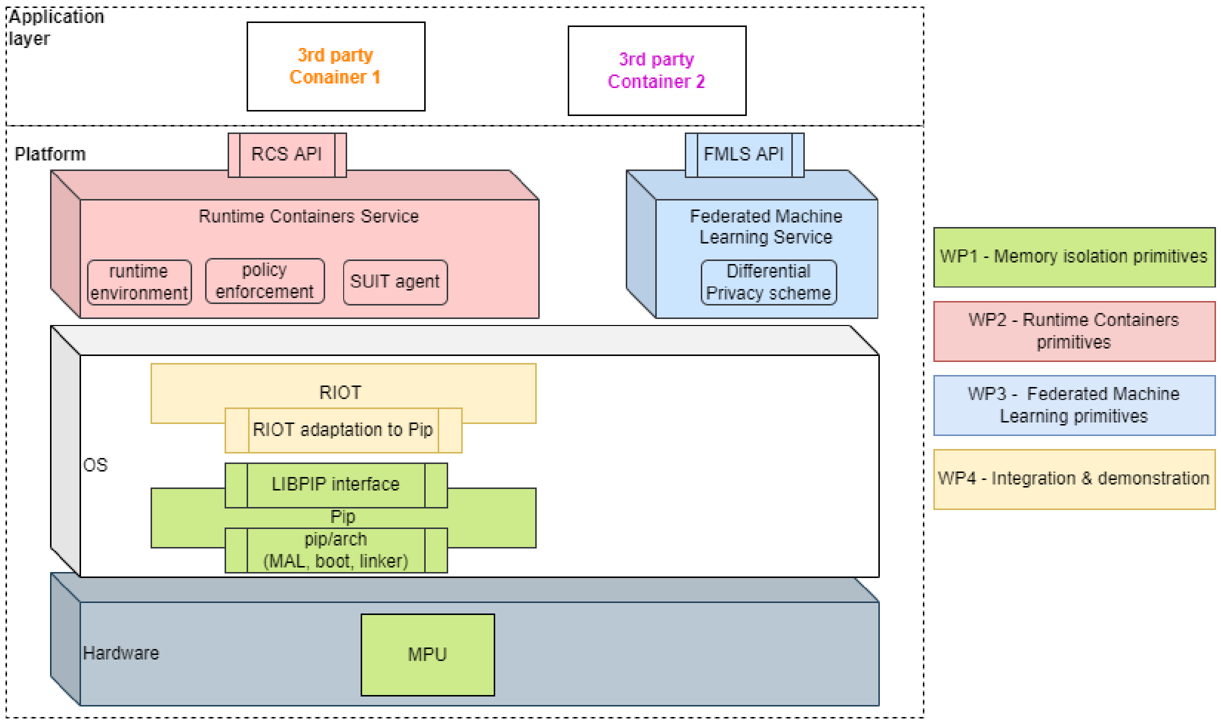
Application layer
3rd party Conainer 1
3rd party Container 2
Platform
RCS API
FMLS API
Runtime Containers Service
runtime environment
policy enforcement
SUIT agent
Federated Machine Learning Service
Differential Privacy scheme
OS
RIOT
RIOT adaptation to Pip
LIBPIP interface
Pip
pip/arch (MAL, boot, linker)
Hardware
MPU
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 2: High-Level Architecture including all components

1.1 Modular configurations of the architecture

We propose a modular architecture that allows developers to either use TinyPART building blocks separately or conjointly. The benefits of each use case for specific use cases will be investigated in deliverable D4.3.1. The different configurations possible are the following:

- Individual building blocks
  - Configuration C corresponds to the Runtime Container and is depicted in figure 3.
  - Configuration D corresponds to the Federated Machine Learning Service, as depicted in 4
  - Configuration P corresponds to Pip, the memory isolation solution, as depicted in 5
- Paired building blocks
  - Configuration CD, depicted in 6, corresponds to the combination of the Runtime Container Service and the Federated Machine Learning Service.
  - Configuration PC, depicted in figure 7, consists in combining Pip and the Runtime Container Service.
  - Configuration PD, depicted in figure 8, corresponds to the combination of Pip and the Federated Machine Learning Service.
- Full integration, depicted in figure 2 corresponds to the combination of all three TinyPART enablers Pip, Runtime Containers Service and Federated Machine Learning Service.

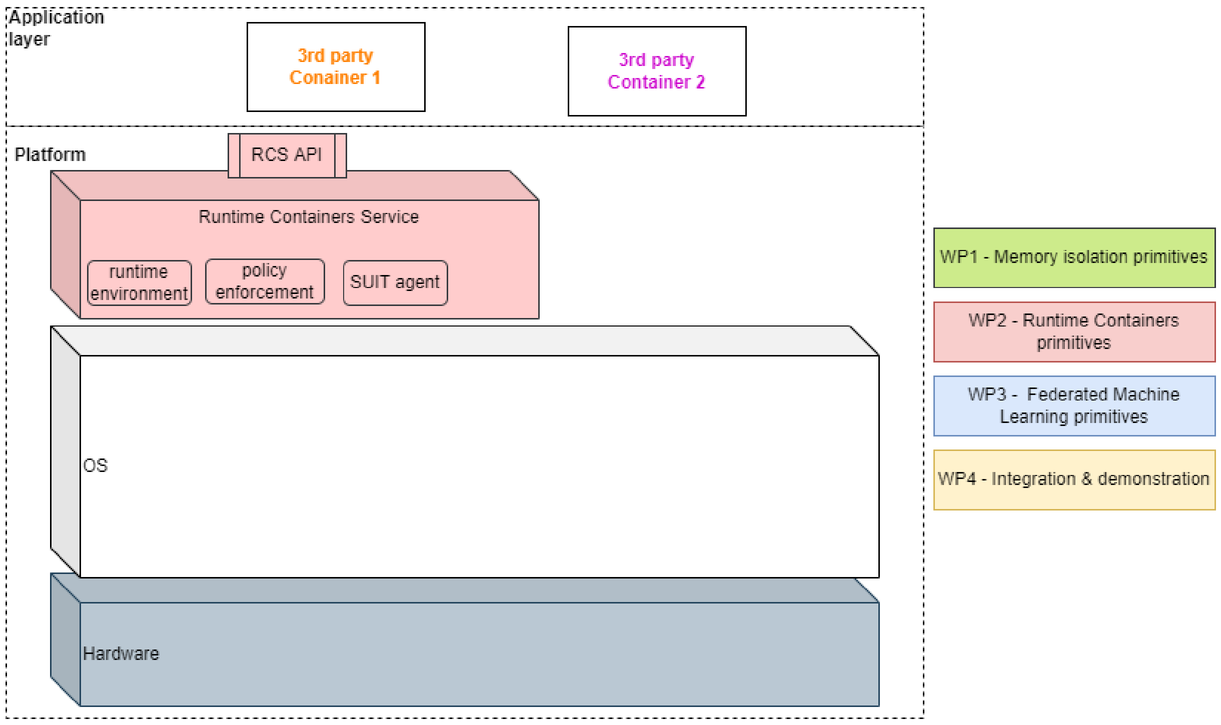
Application layer
3rd party Conainer 1
3rd party Container 2
Platform
RCS API
Runtime Containers Service
runtime environment
policy enforcement
SUIT agent
OS
Hardware
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 3: High-Level Architecture Configuration C

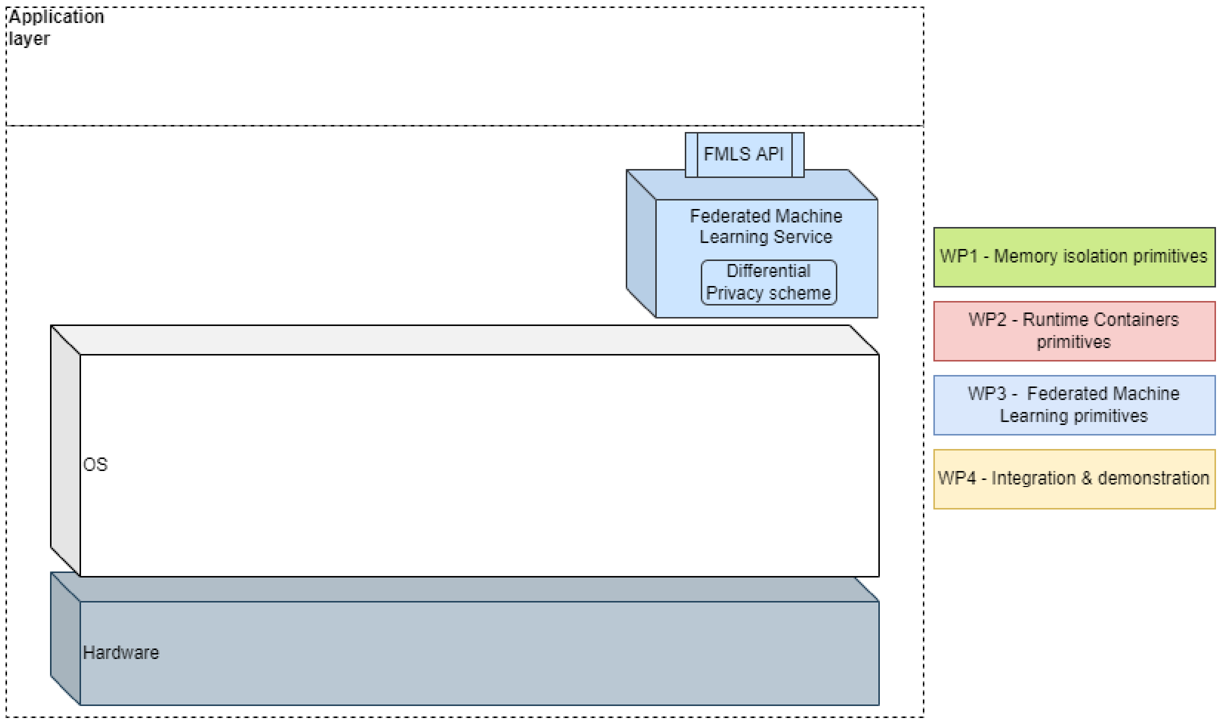
Application layer
FMLS API
Federated Machine Learning Service
Differential Privacy scheme
OS
Hardware
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 4: High-Level Architecture Configuration D

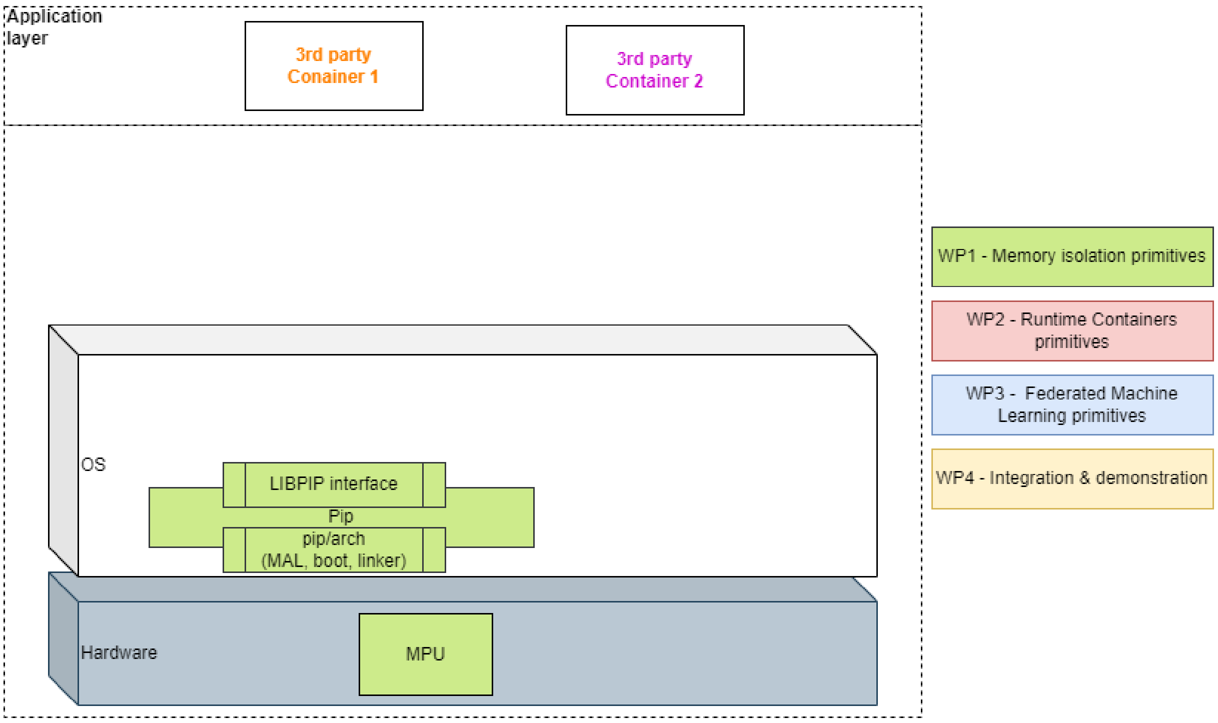
Application layer
3rd party Conainer 1
3rd party Container 2
OS
LIBPIP interface
Pip
pip/arch (MAL, boot, linker)
Hardware
MPU
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 5: High-Level Architecture Configuration P

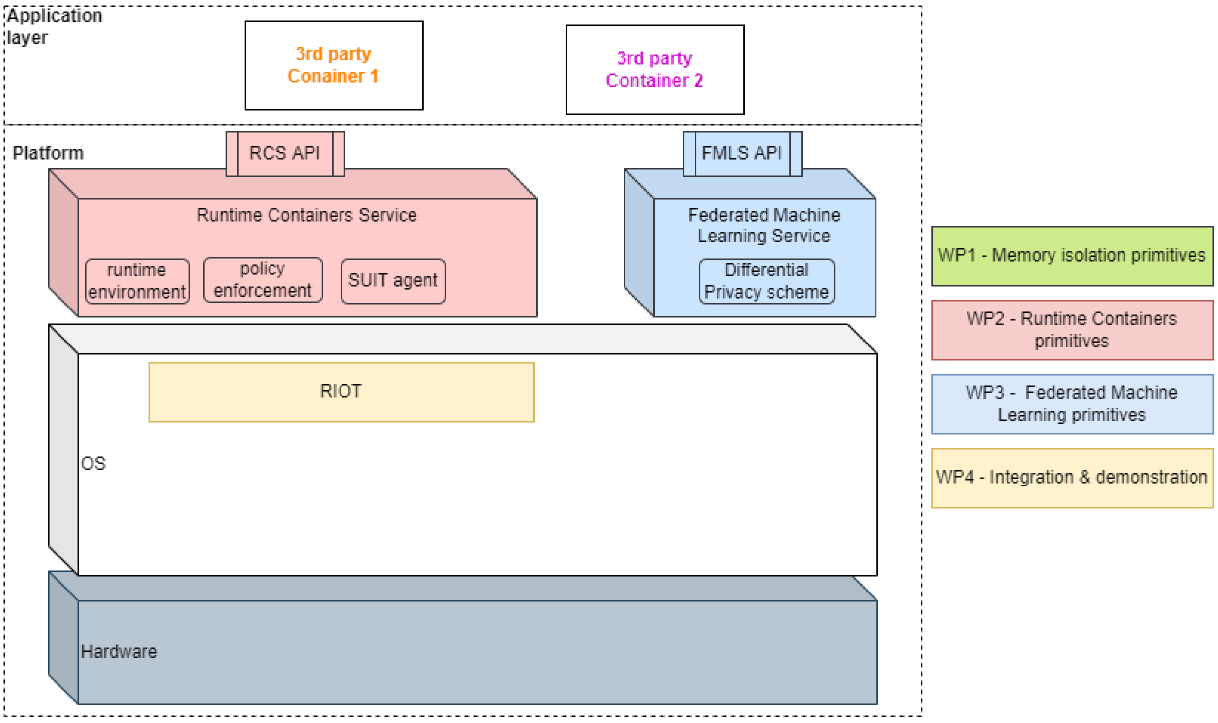
Application layer
3rd party Conainer 1
3rd party Container 2
Platform
RCS API
FMLS API
Runtime Containers Service
runtime environment
policy enforcement
SUIT agent
Federated Machine Learning Service
Differential Privacy scheme
RIOT
OS
Hardware
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 6: High-Level Architecture Configuration CD

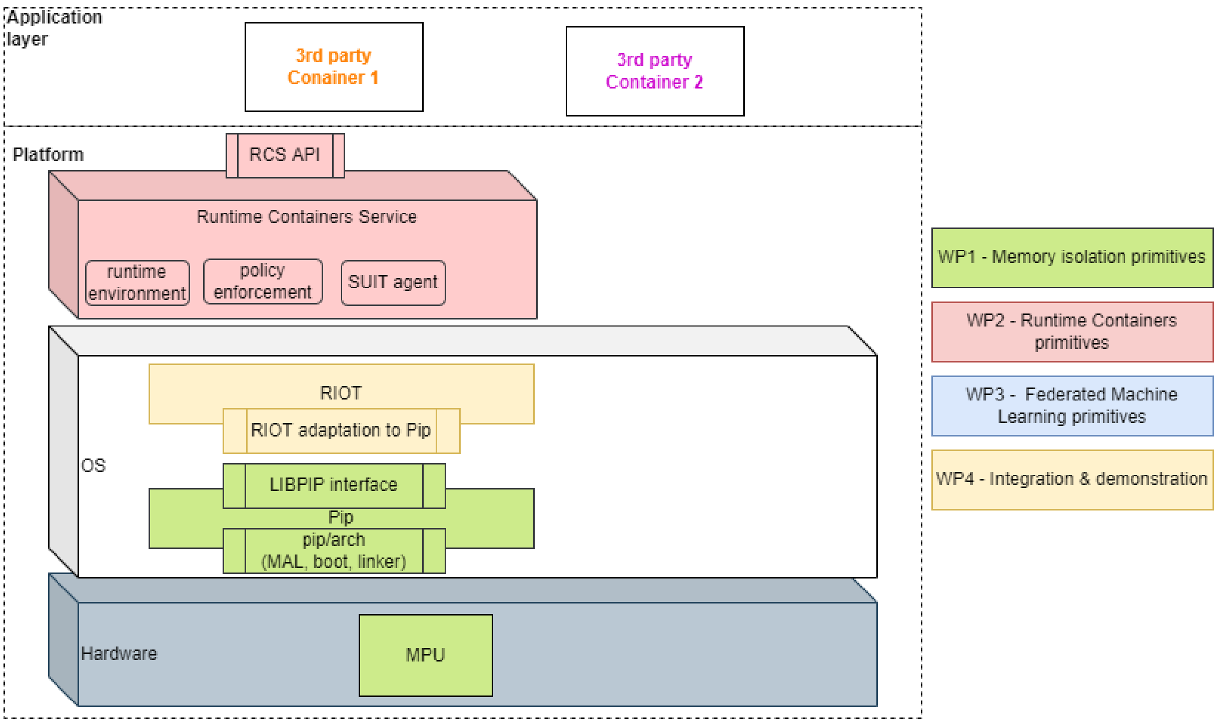
Application layer
3rd party Conainer 1
3rd party Container 2
Platform
RCS API
Runtime Containers Service
runtime environment
policy enforcement
SUIT agent
RIOT
RIOT adaptation to Pip
OS
LIBPIP interface
Pip
pip/arch (MAL, boot, linker)
Hardware
MPU
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 7: High-Level Architecture Configuration PC

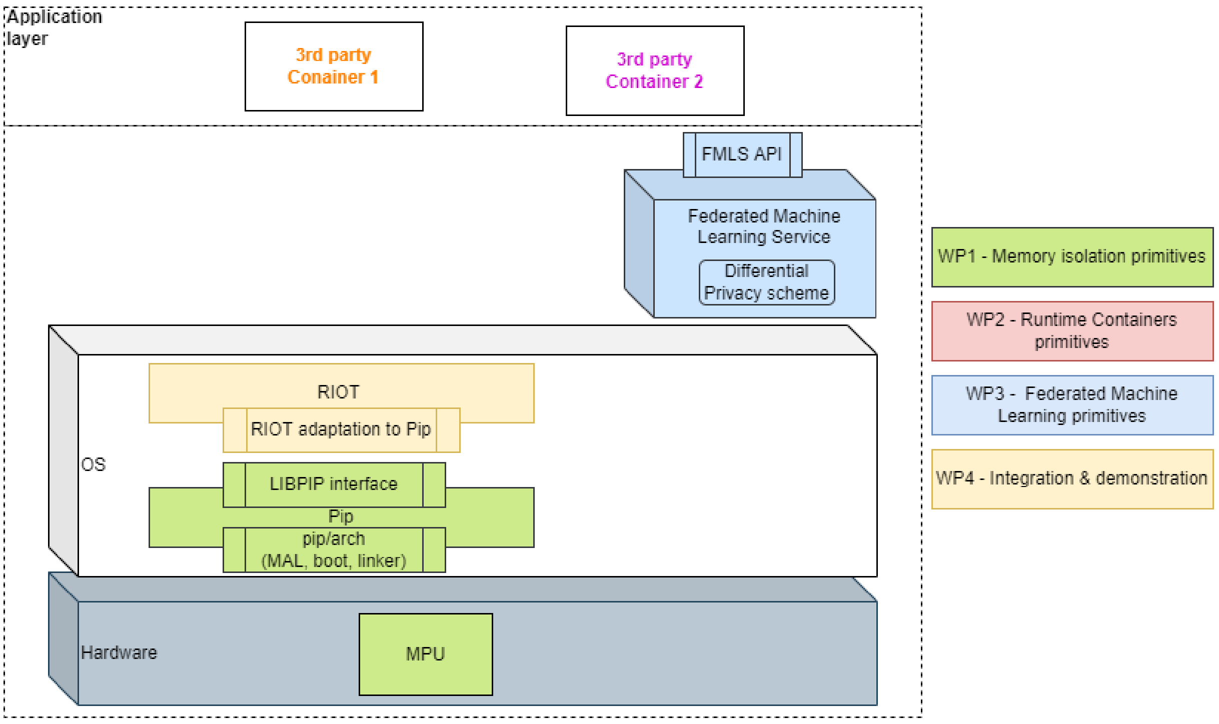
Application layer
3rd party Conainer 1
3rd party Container 2
FMLS API
Federated Machine Learning Service
Differential Privacy scheme
RIOT
RIOT adaptation to Pip
OS
LIBPIP interface
Pip
pip/arch (MAL, boot, linker)
Hardware
MPU
WP1 - Memory isolation primitives
WP2 - Runtime Containers primitives
WP3 - Federated Machine Learning primitives
WP4 - Integration & demonstration

Figure 8: High-Level Architecture Configuration PD

1.2 Development strategy

We propose to follow the iterative development strategy depicted in figure 9 to minimize development and integration risks and to achieve the highest scientific value for the project.

First, we will investigate the scientific challenges in the individual building blocks in order to achieve the project's Minimal Viable Product [1].

Afterwards, we will integrate modules two by two, starting by the configuration CD and the integration of RIOT over PIP. After this, depending on the remaining time available in the project, we will aim for full integration and/or document the residual steps to be taken to finalize the integration of components.

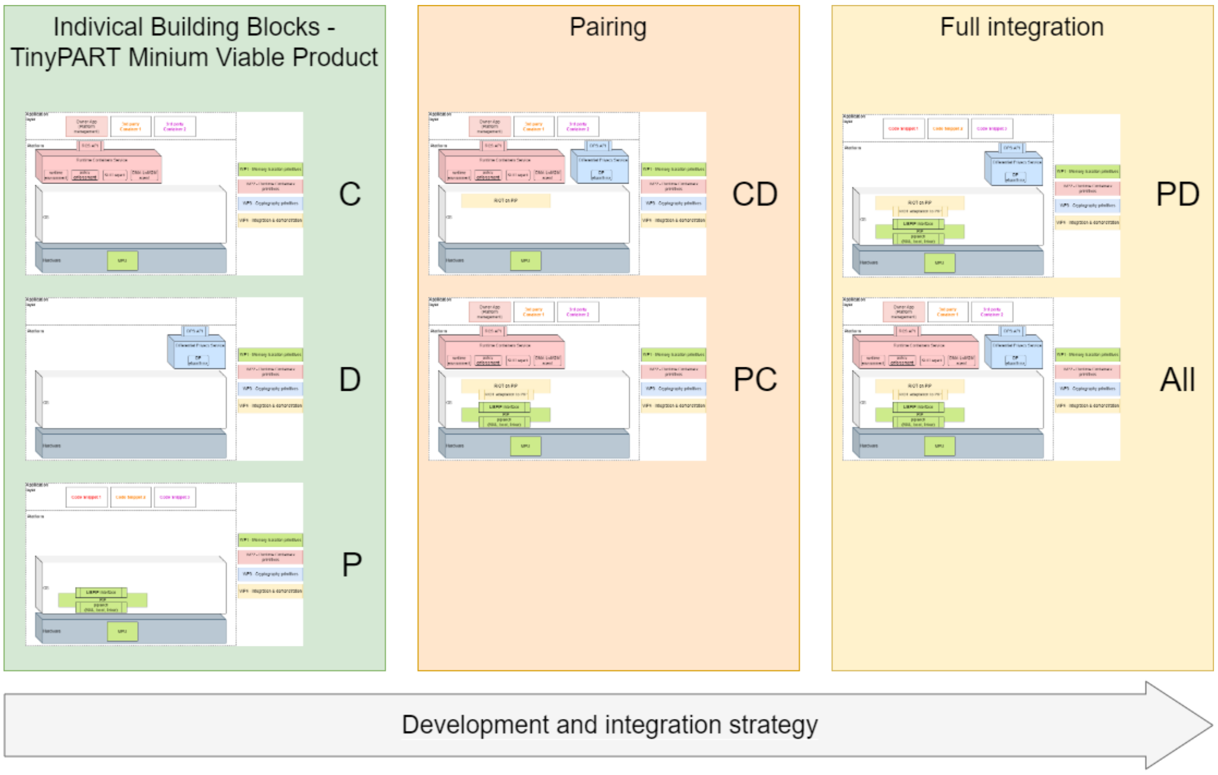
Indicival Building Blocks -
TinyPART Minium Viable Product
Pairing
Full integration
C
D
P
CD
PC
PD
All
Development and integration strategy

Figure 9: Development and integration strategy

[1] A minimum viable product, or MVP, is a product with enough features to attract early-adopter customers and validate a product idea early in the product development cycle.

2 Description of modules

2.1 PIP

Pip is an OS (Operating System) kernel specialised in memory management and context switching. It is a partitioning kernel and a security kernel, ensuring memory isolation between partitions. It is part of the proto-kernel family, *e.g.* leaving file systems, drivers, scheduler, IPC (Inter-Process Communication) or multiplexer out of the minimalist TCB (Trusted Computing Base). Therefore, Pip is sole privileged in the system while all the partitions run unprivileged in userland (see Figure 10). The partitions provide the missing features when necessary (for instance scheduling) as well as the applications and their support components. Pip is meant to be trustworthy and proofs of its memory isolation is on going work.

In the rest of the document, we speak about Pip-MPU, the Pip variant which memory isolation is based on the MPU (Memory Protection Unit, see Section 2.1.6 for some insights) and targets low-end microcontrollers. The informed reader already acquainted with the MMU-based memory isolation variant (the first Pip release [1]) is highly recommended to read the Pip-related parts in this document as no comparison or evolution from the MMU variant is described. Pip-MPU is intended to be compatible with both ARMv7-M and ARMv8-M architectures.

2.1.1 Partitioning model

Pip's memory management is based on a hierarchical partitioning model. The main principle is that a *partition* (an execution unit) can create one or several sub-partitions, that in turn can create sub-partitions. This creates a *partition tree* as can be seen in Figure 10, rooted in a special partition called the *root partition*. The root partition is the only partition existing at system initialisation. The other partitions are dynamically created during the system's lifetime.

2.1.2 Partition lineage

We define in this paragraph fundamental relationships within the partition tree, closely resembling a family tree. The relationship between a partition and a sub-partition is referred as a **parent-child** relationship. A **parent partition** may have **descendants** when a **child partition** also have its own children. For the descendants, this parent partition is considered as an **ancestor**. The common ancestor to all partitions is the **root partition**. Partitions stemming from the same parent are **sibling partitions**.

2.1.3 Features

Pip exposes system calls that build up the partition tree. The system calls are detailed in Section 3.1.

Memory sharing A parent partition can create child partitions and share memory with them. All the memory attributed to a partition is directly accessible to this partition. The parent can only map memory that it owns itself. As a result, the whole system memory is divided in two parts: 1) Pip's memory and 2) the rest owned at start by the root partition. The root partition can access any partition's memory. This is a natural consequence of the hierarchical partitioning model, where the security of any partition is based on its ancestors, by recursion. As such, a parent partition has always the ability to tear down a child.

Access permissions The parent decides which access permissions is associated to the shared memory. Indeed, a partition can restrict the use of the memory to a child partition. However, it can never increase the permissions.

Fine-grained memory unit The basic memory unit is a memory block, *i.e.* a continuous range of memory addresses. The memory block is defined by a start and an end address and its size can vary during the partition's lifetime. Indeed, the user can cut memory blocks in smaller pieces, for example to share only a part of a block to a child.

Flexible usage Partitions can leverage the previous features to set up their own requirements and fit their usage. Examples are inter-partition communication by moving around memory blocks from isolated partitions, ensuring the `W^X` principle and least privilege in a partition... From the start of the system with the launch of the root partition, the partitioning view is completely dynamic and evolves as desired by the user.

2.1.4 Security properties

Pip's partitioning model ensures 3 security properties:

- **Vertical sharing**: a parent partition can only share memory to a child partition and must own the memory blocks it gives out.
- **Horizontal isolation**: a memory block can only be shared to as single child partition. This property ensures strict memory isolation between child partitions.
- **Kernel Isolation**: Pip's memory, as well as all metadata structures used for describing the partitioning model, should never be accessible to any partitions. This property protects the kernel itself and ensures no partition can change its configuration by itself by bypassing Pip.

As such, the Pip API builds up a nested compartmentalisation. Pip-MPU ensures these security properties are guaranteed at any time.

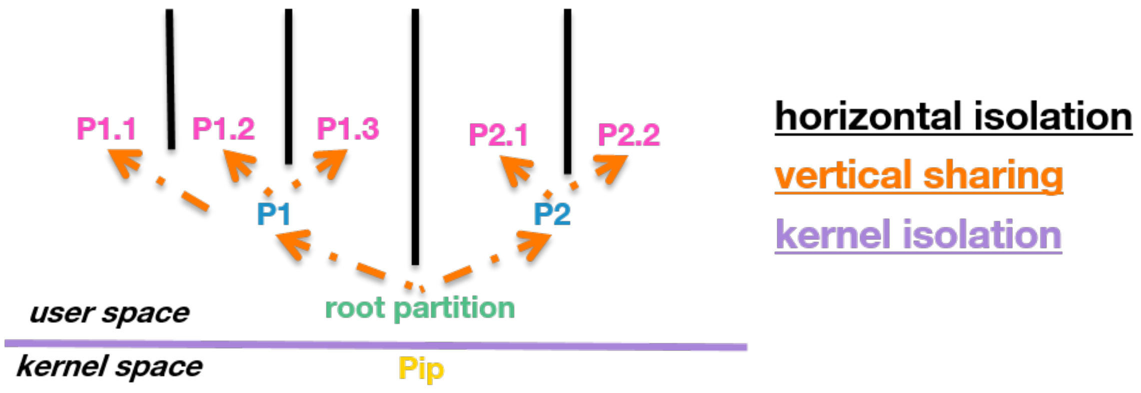

Figure 10: Pip's partitioning scheme.

2.1.5 Pip nomenclature

This paragraph aims at defining some Pip objects that are used to describe the API in Section 3.1. This is a simplified view hiding the implementation detailed not relevant for the user.
The metadata structures are Pip's configuration information, used to track the partitioning view and to configure the MPU with the correct information. In Pip, they are of two types :

1. **Partition Descriptor**: A partition is identified by a Partition Descriptor, unique in the system for this partition. The Partition Descriptor is a metadata structure containing the partition's references.
2. **Kernel structure**: The kernel structure is a metadata structure holding the memory blocks and associated RWX access permissions (Read-Write-Execute). A kernel structure has a fixed amount of slots (user-defined, by default 8) where to register the memory blocks. Hence, a partition can have several kernel structures. If one needs more slots to register new memory blocks, it is sufficient to create new kernel structures, up to a user-defined limit.

Metadata structures are created out of memory blocks. They are only special memory blocks containing Pip information. As such, memory blocks that turned into metadata structures are not accessible anymore to the partition, as they became Pip objects and should be isolated (Kernel Isolation property).

2.1.6 Focus on the MPU management

The MPU (Memory Protection Unit) [2] is a hardware unit present in many ARM Cortex-M processors. Its role is to restrict memory accesses according to a defined configuration. The configuration consists in a set of memory regions called *MPU regions* having various permission rights (read, write, execute) and additional attributes (caching, buffering...). From a broad perspective, the MPU is for small embedded systems what the Memory Management Unit (MMU) is for general-purpose systems, since both share the memory protection role in a similar fashion.

A Pip-based system is guarded by the MPU at all time. The memory blocks are meant to be configured as an MPU region, with a minimum size of 32 bytes. This means an illegal access (to Pip or a violation of a block's access permissions) ends up in a memory fault. The memory fault is caught by the parent partition which can decide what to do with the faulted child.

Furthermore, Pip introduces a distinction between all memory blocks owned by a partition (the block pool) and the memory blocks that can be accessed (enabled blocks). Indeed, a typical MPU consists in 8 protected MPU regions that are too restrictive for the partition management. Not all memory blocks can thus be active at the same time. Thus, blocks in the block pool are enabled or disabled. **Enabled** means they are guarded by the MPU with their RWX permissions set up. **Disabled** means they are not accessible at all. **Trying to access disabled blocks, even if they are in the block pool, results in a memory fault.** By default, exception made for the root partition, no block is active from the block pool and needs to be manually enabled by the developer (see Section 3.1 for usage). Though, enabled blocks must not be confused with **accessible** blocks. In fact, when a memory block is shared, it is accessible by default, *i.e.* eligible to be enabled, and the developer choose to enable it or not. However, if it leaves this shared state condition, *e.g.* by turning into a metadata structure during partition creation, it becomes inaccessible. Only accessible blocks can be enabled in the MPU.

2.1.7 Special consideration to ARMv7-M users

The ARMv8-M architecture introduces slight differences with the MPU of the previous ARMv7-M version. Indeed, ARMv7-M requires more configuration constraints. Pip developed mechanisms to deal with these constraints so to share common grounds between the two versions. Basically, this results in 2 consequences to the ARMv7-M user:

- Stack size and alignment: the stack must always be aligned to a power of two aligned on a multiple of its size. This is a direct constraint carried forward.
- Access to non-aligned memory blocks: Pip releases the alignment constraint to any other memory blocks. However, in the background, Pip makes partial memory block configuration that matches the alignment constraint. Hence, for any legitimate memory access in an enabled memory block, Pip reloads the MPU to always match the constraints. This means that a legitimate access to a non-aligned memory block could be delayed because of the MPU reloading.

2.2 LIBPIP

@ Damien: Proofread

LIBPIP is a C library that provides PIP-related operations to the operating system. Its API fit one by one with the PIP interface. The operating system may use LIBPIP to create and manage child partitions, for context switching between partitions, to configure interrupt handlers, or to access PIP controlled ressources - typically some registers or operations that required privileged mode.
Each child partition should integrate LIBPIP. Nevertheless, a partition which doesn't need the full API may use a light version of LIBPIP that only embeds the required functions.

2.3 RIOT

RIOT is an open source low-power OS designed to run on heterogeneous IoT microcontrollers typically found in the Internet of Things (Arm Cortex-M, RISC-V etc.).

Table 2: Overview of common system APIs to implement applications in RIOT.

Core APIs	Hardware Abstraction APIs	Timer API	High-level Network APIs

`thread`
thread handling
`msg, mbox, thread_flags`
inter-process comm.
`mutex`, `semaphore`
concurrency handling
	`periph/uart, spi, ...`
	access to MCU peripherals
	`SAUL`
	access to sensors/actuators
	`netdev`
	access to network devices
	`mtd`
	access to storage devices
		`ztimer, xtimer`
		high-level timer abstraction
			`sock`
			generic interface to high-level connectivity
			`POSIX socket`
			based on sock

2.3.1 RIOT Architecture

RIOT is structured in software modules that are aggregated at compile time, around a kernel providing minimalistic functionality. This approach allows to build the complete system in a modular manner, including only modules that are required by the use-case. This not only minimizes memory consumption, but also system complexity in the field.
At a high level, the RIOT code is structured according to the groups listed below:

- `core` implements the kernel and its basic data structures such as linked lists, LIFOs, and ringbuffers;
- hardware abstraction distinguishes four parts: (i) `cpu` which implements functionalities related to the microcontroller, (ii) `boards` which mostly selects, configures, and maps the used CPU and drivers, (iii) `drivers` which implements device drivers, and (iv) `periph` which provides unified access to microcontroller peripherals and is used by device drivers;
- `sys` implements system libraries beyond the functionalities of the kernel, such as crypto functionalities, file system support and networking;
- `pkg` imports third-party components (libraries which are not included in the main code repository);
- `application` implements the high-level logic of the actual use-case.

Within code groups, functionalities are split into modules, each of which consists of one or more API header file(s) in the include path, and a single directory containing all its code. Modules may have sub-modules used for tailoring functionalities at a finer grain.
It is worth noting that the RIOT structure naturally factors out high-level application logic. Thus, a straightforward approach (and current best practice) is to develop and maintain high-level application code in a separate repository which typically pulls in the particular RIOT release the application is built upon.
The minimal configuration bundles only the `core` module (without any sub-modules), a `cpu` module and a `board` module. Every other module is optional.

2.3.2 RIOT Application Programming Interfaces

This section recaps the main application programming interfaces defined by RIOT. These interfaces are consistent across all hardware supported by RIOT, meaning that coding against such interfaces guarantees cross-platform code portability. See Table 2, which lists the APIs, as well as which shows how these APIs articulate. The APIs fall into four categories:

Core APIs provide basic OS kernel interfaces to manage threads, interprocess communication, and concurrency.

Hardware abstraction APIs provide generic interfaces for abstracting MCU peripherals as well as unified access to device drivers grouped by device classes;

Timer APIs provide unified high-level access to the configured, platform dependent timer infrastructure;

Network APIs provide a generic interface to different network stacks.

2.3.3 RIOT over PIP-MPU

@ Damien: Proofread

RIOT has been designed to run as the main operating system of a microcontroller. Notably, RIOT assumes having full access to the system and being run in privileged mode. On its side, the PIP hypervisor only allows unprivileged thread mode and controls access to some ressources. Also, running RIOT over PIP-MPU required some adaptations:

- LIBPIP is an external library. To integrate and adapt the library with RIOT, a new RIOT package is required.
- The booting and the scheduling code of RIOT have to be adapted.
- Each modules that required access to sensible registers or operations has to be adapted.
- We have choosen to create new dedicated cpu and board target in RIOT to specifically support PIP-MPU.

2.4 Runtime Containers Service

The Runtime Containers Service is composed by three modules, namely the runtime environment, the policy enforcement, Software Update agent (either SUIT or OMA-LwM2M) which manage different aspects of Containers. This section provides an overview of each of these components.

2.4.1 Runtime environment

The runtime environment plays the role of a middleware application for the scripts in the containers thanks to the RCS API, described in section 3.2, which exposes the methods of the device.

The runtime environment retrieves the script metadata which contain indication on the resources required by the script and creates an execution context. Some example of script metadata are the communication endpoints required (e.g., serial, BLE), a mask indicating which methods exposed by the device are required by the Container and authorized by the device's owner, the list of methods and variables exposed by the container and that are authorized to be used by other applications or containers. These resources exposed by either the Container or the Device can be related to private data, thus it is essential to manage the access to either the container or device ressources. To this end, the execution context can be used to set up and enforce a policy enforcement module (described in section 2.4.2) which manages both types of resources.

2.4.2 Policy enforcement

The policy enforcement module is a security layer which participates in the isolation mechanisms to guarantee that only authorized actors have access to only the data they are supposed to access. It enforces some policies that are decided at the level of the Script & Device Management Platform. Therefore, it is necessary that this module and the Script & Device Management Platform agree on some algorithms and secrets to communicate.

The device resources required for a given script are included in the script metadata by the Script & Device Management Platform. The policy enforcement module can retrieve and verify these metadata to configure an internal security policy and enforce it. For example, these rights can be expressed in the form of a mask indicating which device resources are required by the script. Upon reception of the script and the associated mask, the policy enforcement module can ensure that each of the subsequent requests from the container only target the authorized device resources.

2.4.3 Software Update Agent : SUIT Agent

A commissioning phase provisions the IoT device with a bootloader and two RIOT firmware slots, one of which is loaded with an initial operating system (RIOT) image including a Software Update Agent which sits on top of the network stack and which features the necessary cryptographic material: the public key of the authorized software maintainer, according to the SUIT standard (i.e. specified in draft-ietf-suit-manifest, which we co-author at IETF).
Next, software udpate and maintenance life-cyles can take place on the IoT device, over the low-power network, in compliance with the SUIT specifications.
The authorized software maintainer can:

1. build a RIOT software update (firmware update or runtime container update),
2. produce the corresponding SUIT manifest, i.e. metadata and cryptographic material,
3. PUT the SUIT manifest and the software update to the software update server.

In turn, the IoT device can:

1. poll the software update server for available software updates,
2. GET and verify SUIT manifests,
3. GET and validate software updates, installs/reboots if validated (else: sends an notification).

As such, the Software Update Agent ensures end-to-end security for software updates on low-power IoT devices, providing strong cryptographic guarantees, for example, based on ed25519 digital signatures and SHA256 hashing – the use of alternative cryptographic primitives is also possible. By end-to-end, we here mean from the authorized software maintainer to the IoT device itself.
Using the mechanisms specified by SUIT, the prototype we demo can for example mitigate the below attacks.

Mitigating Tampered Software Update Attacks An attacker may try to update the IoT device with a modified and intentionally flawed software image. To counter this threat, our prototype based on SUIT uses digital signatures on a hash of the image binary and the metadata to ensure integrity of both the software and its metadata.

Mitigating Unauthorized Software Update Attacks An unauthorized party may attempt to update the IoT device with modified image. Using digital signatures and public key cryptography, our prototype based on SUIT ensure that only the authorized maintainer (holding the authorized private key) will be able to update de device.

Mitigating Software Update Replay Attacks An attacker may try to replay a valid, but old (known-to-be-flawed) software. This threat is mitigated by using a sequence number. Our prototype based on SUIT uses a sequence number, which is increased with every new software update.

Mitigating Software Update Mismatch Attacks An attacker may try replaying a software update that is authentic, but for an incompatible device. Our prototype based on SUIT includes device-specific conditions, which can be verified before installing a software image, thereby preventing the device from using an incompatible software image.

2.5 Federated Machine Learning Service

The objective of this module is to provide Federated Learning (FL) primitives to the containers. FL involves training of machine learning model parameters on-board on local client IoT devices using their own local datasets and sharing these trained parameters to a global entity for aggregation. For next round of training, local models are re-initialized to global model parameters aggregated at previous round and local training again happens on these local clients. Following that aggregation process is carried out on global server. This training cycle repeats until convergence which fulfils performance on test dataset on these different clients.

A Differential Privacy scheme is encapsulated in the Federated Learning Service. Its objective is to ensure privacy for exchange of these trained parameters between clients and server, only gradients of these model parameters are shared after clipping its values and with addition of Gaussian noise during stochastic gradient descent update on the client side as covered in [3].

3 Description of interfaces

3.1 PIP services

This section presents Pip's memory management interface. The remaining part of the API that covers context switching is in progress and will be released later on.

Pip exposes 11 system calls that are callable by the userland partitions. They target either a child partition or the current partition:

1. `Pip_createPartition`: creates a child partition to the current partition.
2. `Pip_deletePartition`: deletes a child partition.
3. `Pip_prepare`: sets up a new kernel structure to the current partition's or a child partition.
4. `Pip_collect`: finds an empty kernel structure in the current partition or in a child partition and removes it.
5. `Pip_addMemoryBlock`: adds a block (shared memory) to a child partition (consumes one slot in the kernel structure).
6. `Pip_removeMemoryblock`: removes a block from a child partition (frees one slot in the kernel structure).
7. `Pip_cutMemoryblock`: cuts a block in the current partition (consumes one slot in the kernel structure).
8. `Pip_mergeMemoryBlocks`: merges back previously cut blocks in the current partition (frees one slot in the kernel structure).
9. `Pip_mapMPU`: selects a block to be enabled in the MPU of the current partition or a child partition.
10. `Pip_readMPU`: reads the block mapped in a MPU entry of the current partition or a child partition.
11. `Pip_findBlock`: finds a block in the current partition or in a child partition.

In the following, *blockLocalId* refers to the block's local id (as referenced in the currently activated partition). Furthermore, as the block containing the partition's Partition Descriptor is unique in the whole system, we refer to this block or the Partition Descriptor indifferently.

3.1.1 `Pip_createPartition`

Prototype	`uint32_t Pip_createPartition(uint32_t *blockLocalId)`
Description	Creates a child partition to the current partition. The block <blockLocalId> becomes a Pip object and inaccessible to the partitions.
Parameter(s)	• blockLocalId: The block that will contain the child's Partition Descriptor structure.
Return value	1:OK/0:NOK.

Example:

```
uint32_t *childPartDescBlockLocalId = argv[0]; // Retrieve block id from stack
if (!Pip_createPartition(childPartDescBlockLocalId))
{
  printf("Failed to create childPartDescBlockLocalId...\n");
  for (;;);
}
```

3.1.2 `Pip_deletePartition`

Prototype	`uint32_t Pip_deletePartition(uint32_t *childPartDescBlockLocalId)`
Description	Deletes the child partition <childPartDescBlockLocalId> and gives back all its blocks to the current partition. All the child's descendants are deleted, as well as their respective blocks, that are all accessible to the ancestors again.
Parameter(s)	• childPartDescBlockLocalId: The block containing the Partition Descriptor structure of the child partition to delete.
Return value	1:OK/0:NOK

Example:

```
if (!Pip_deletePartition(childPartDescBlockLocalId))
{
  printf("Failed to delete childPartDescBlockLocalId...\n");
  for (;;);
}
```

3.1.3 `Pip_prepare`

Prototype	`uint32_t Pip_prepare(uint32_t *partDescBlockId, int32_t projectedSlotsNb, uint32_t *requisitionedBlockLocalId)`
Description	Sets up a kernel structure for the partition <partDescBlockId>. The latter could be the current partition or a child partition. The block <requisitionedBlockLocalId> becomes a Pip object and inaccessible to any partitions.
Parameter(s)	• partDescBlockId: the block containing the Partition Descriptor of the partition to prepare (current partition or a child partition). • projectedSlotsNb: the number of requested slots. The value "-1" forces the *prepare*, even if there are still free slots. • requisitionedBlockLocalId: the block that will contain the new kernel structure.
Return value	1:OK/0:NOK

Example:

```
uint32_t *rootKernStructBlockLocalId = argv[1]; // Retrieve block id from stack
if (!Pip_prepare(childPartDescBlockLocalId, -1, rootKernStructBlockLocalId))
{
  printf("Failed to prepare childPartDescBlockLocalId...\n");
  for (;;);
}
```

3.1.4 `Pip_collect`

Prototype	`uint32_t *Pip_collect(uint32_t *partDescBlockId)`
Description	Collects an empty kernel structure, if possible, from the partition <partDescBlockId>. The latter could be the current partition or a child partition. The retrieved block, if any, becomes accessible again to all partitions. The partition <partDescBlockId> must have been previously prepared via the `Pip_prepare` system call.
Parameter(s)	• idPDchild: the block containing the Partition Descriptor of the partition to collect (current partition or a child partition).
Return value	the block id containing the collected empty structure/NULL otherwise

Example:

```
if (!Pip_collect(childPartDescBlockLocalId))
{
  printf("Failed to collect from childPartDescBlockLocalId...\n");
  for (;;);
}
```

3.1.5 `Pip_addMemoryBlock`

Prototype	`uint32_t* Pip_addMemoryBlock(uint32_t *childPartDescBlockLocalId, uint32_t *blockToShareLocalId, uint32_t r, uint32_t w, uint32_t e)`
Description	Adds the block <blockToShareLocalId> to the child partition <childPartDescBlock-LocalId>. The block is still accessible to the current (parent) partition acting as shared memory. In the child partition, the block gets the <r w e> access rights.
Parameter(s)	• childPartDescBlockLocalId: the block containing the child partition's Partition Descriptor • idBlockToShare: the block local id to share with the child partition • r w e: the rights to apply to the shared block in the child partition
Return value	Returns the shared block's id in the child/NULL otherwise

Example:

```
uint32_t *childVidtBlockLocalId = argv[2]; // Retrieve block id from stack
uint32_t *childVidtBlockIdInChild = Pip_addMemoryBlock(childPartDescBlockLocalId,
  childVidtBlockLocalId, 1, 1, 0); // RW

if (childVidtBlockIdInChild == NULL)
{
  printf("Failed to add block for childVidtBlockLocalId...\n");
  for (;;);
}
```

3.1.6 `Pip_removeMemoryBlock`

Prototype	`uint32_t Pip_removeMemoryBlock(uint32_t *blockToRemoveLocalId)`
Description	Removes the block <blockToRemoveLocalId> from the child partition where it has been previously added. As Pip enforces the “Horizontal Isolation” property, the block can only be found in a single child partition. The block <blockToRemoveLocalId> must have been previously added via the `Pip_addMemoryBlock` system call. This operation fails if the entire block can’t be considered as shared memory anymore, in particular: • if the child or its descendants used it (or part of it) as a kernel structure (for instance, after a cutMemoryBlock and createPartition) • if the child has a descendant that has cut the block • if the child shared a part of it (for instance, after a `Pip_cutMemoryBlock` and `Pip_addMemoryBlock`)
Parameter(s)	• blockToRemoveLocalId: the block local id to remove from the child partition
Return value	1:OK/0:NOK

Example:

```
if (!Pip_removeMemoryBlock(blockToRemoveLocalId))
{
  printf("Failed to remove block blockToRemoveLocalId...\n");
  for (;;);
}
```

3.1.7 `Pip_cutMemoryBlock`

Prototype	`uint32_t *Pip_cutMemoryBlock(uint32_t *blockToCutLocalId, uint32_t *cutAddr, int32_t mpuRegionNb)`
Description	Cuts the block <blockToCutLocalId> at address <cutAddr>, *i.e.* the block is shortened with a new block end address at <cutAddr -1>. A new sub-block is created spanning from <cutAddr> to the block’s original end address. The created sub-block is added to the blocks list. It can also be configured in the physical MPU if the region number is valid.
Parameter(s)	• blockToCutLocalId: the block to cut • cutAddr: the cut address, it is the created sub-block’s start address • mpuRegionNb: If valid, the physical MPU’s region number that will protect the new sub-block. If not valid, the sub-block is still created but not configured in the MPU.
Return value	the created sub-block local id/NULL otherwise

Example:

```
// Cut the block <blockToShareLocalId> at address 0x20002000 and configure the newly created
   sub−block in MPU region number 3
uint32_t *childKernStructBlockLocalId = Pip_cutMemoryBlock(blockToShareLocalId, (uint32_t *)
   0x20002000, 3);

if (childKernStructBlockLocalId == NULL)
{
    printf("Failed to cut block blockToShareLocalId...\n");
    for (;;);
  }
```

3.1.8 `Pip_mergeMemoryBlocks`

Prototype	`uint32_t *Pip_mergeMemoryBlocks(uint32_t *blockToMerge1LocalId, uint32_t *blockToMerge2LocalId, int32_t mpuRegionNb)`
Description	Merges blocks <blockToMerge1LocalId> and <blockToMerge2LocalId> together. The block <blockToMerge1LocalId> must have been previously cut via the `Pip_cutMemoryBlock` system call. The two sub-blocks are are removed from the physical MPU if they were configured. Instead, the merged block can be directly configured in the physical MPU if the region number is valid.
Parameter(s)	• blockToMerge1LocalId: the first sub-block's id that corresponds to the first part of the final merged block. • blockToMerge2LocalId: the second sub-block's id that corresponds to the second part of the final merged block. • mpuRegionNb: If valid, the physical MPU's region number that will protect the merged block. If not valid, the merged block is still created but not configured in the MPU.
Return value	the merged block's local id/NULL otherwise

Example:

```
  // Merge blockToShareLocalId with childKernStructBlockLocalId  and configure the merged
    block in MPU region number 3
  if (!Pip_mergeMemoryBlocks(blockToShareLocalId, childKernStructBlockLocalId, 3);)
  {
    printf("Failed to merge sub-blocks blockToShareLocalId with childKernStructBlockLocalId
    ...\n");
    for (;;);
  }
```

3.1.9 `Pip_mapMPU`

Prototype	`uint32_t Pip_mapMPU(uint32_t *partDescBlockId, uint32_t *blockToMapLocalId, int32_t mpuRegionNb)`
Description	Configures the physical MPU region number <mpuRegionNb> with the attributes of block <blockToMapLocalId>, *i.e.* enables a block in the MPU. The system call operates on the current partition or a child partition <partDescBlockId>.
Parameter(s)	• partDescBlockId: the block containing the partition's Partition Descriptor (current partition or a child partition). • blockToMapLocalId: the local block id to map in the MPU. It is local to the partition <partDescBlockId>. • mpuRegionNb: the physical MPU's region number that will protect the block
Return value	1:OK/0:NOK

Example:

```
// Map blockToShareLocalId to childPartDescBlockLocalId's MPU at region number 1
if (!Pip_mapMPU(childPartDescBlockLocalId, blockToShareLocalId, 1))
{
  printf("Failed to map blockToShareLocalId in childPartDescBlockLocalId's MPU configuration
  ...\n");
  for (;;);
}
```

3.1.10 `Pip_readMPU`

Prototype	`uint32_t *Pip_readMPU( uint32_t *partDescBlockId, int32_t mpuRegionNb)`
Description	Reads the local block id mapped in the physical MPU at region <mpuRegionNb> of partition<partDescBlockId>. The block must have been previously mapped via the `Pip_mapMPU` system call. Like `Pip_mapMPU`, the system call operates on the current partition or a child partition <partDescBlockId>.
Parameter(s)	• partDescBlockId: the block containing the partition's Partition Descriptor (current partition or a child partition). • mpuRegionNb: the physical MPU's region number to read from
Return value:	the block local id in the partition <partDescBlockId>/NULL otherwise

Example:

```
if (!Pip_readMPU(childPartDescBlockLocalId, 1))
{
  printf("Failed to read/find a block mapped in childPartDescBlockLocalId's MPU
  configuration...\n");
  for (;;);
}
```

3.1.11 `Pip_findBlock`

Prototype	`int32_t Pip_findBlock(uint32_t *partDescBlockId, uint32_t *addrInBlock, blockOrError *blockAddr)`
Description	Finds a block which memory range covers address <addrInBlock>. If a block is found, it stores the result in a structure of type *blockOrError* (see Listing 1). The system call operates on the current partition or a child partition <partDescBlockId>.
Parameters	• partDescBlockId: the block containing the partition's Partition Descriptor (current partition or a child partition). • addrInBlock: the address stemming from the block to find • blockAddr: the structure where to store the result
Return value	1:OK/Error value:NOK

```
/** From user_ADT.h **/
/**
 * \struct blockAttr
 * \brief blockAttr structure
 */
typedef struct blockAttr
{
    uint32_t* blockentryaddr    ;   //!< Block's local id
    uint32_t* blockstartaddr    ;   //!< Block's start address
    uint32_t* blockendaddr      ;   //!< Block's end address
    uint32_t read : 1           ;   //!< Read permission bit
    uint32_t write : 1          ;   //!< Write permission bit
    uint32_t exec : 1           ;   //!< Exec permission bit
    uint32_t accessible : 1     ;   //!< Block accessible bit
}__attribute__((packed)) blockAttr_t;


/**
 * \struct  blockOrError
 * \brief   blockOrError union structure
 *          When the block is empty, the error flag is set to -1,
 *          otherwise contains a block's public attributes
 */
typedef union blockOrError_t
{
    int32_t error              ;   //!< Error -1 for an empty block
    blockAttr_t blockAttr       ;   //!< A block's publicly exposed attributes
}__attribute__((packed)) blockOrError;
```

Listing 1: blockOrError structure from user_ADT.h

Example:

```
blockOrError* found_block; // initialise result structure
if (!Pip_findBlock(childPartDescBlockLocalId, (uint32_t *) 0x20002000, found_block))
{
  printf("Failed to find a block ranging over address 0x20002000 ...\n");
  for (;;);
}
```

3.2 Runtime Containers Service API

The Runtime Containers Service API allows an owner application to start or stop the Runtime Container Service. It also exposes platform ressources (e.g. a PIN, a sensor, a communication end-point or a software function) to containers.
Different kinds of ressources can be exposed :

- vendor metadata, to identifier on which board the container is running.
- system functions, to retrieve processor ticks or the time.
- self functions, to get environnement status of the container itself.
- sensor accessors, to get value of board sensors (e.g. processor temperature).
- algorithms, optimized function in native code. Currently, only for getting a random number.
- io functions, to read or write to or from containers endpoints.

```
Here is an example of how such resources can be made available :
    uint32_t vendor_get_board_id();    /* each supported board has a unique id             */
    uint32_t vendor_get_vendor_id;     /* non nul if the vendor is registered              */
    uint32_t vendor_get_device_uid;    /* a unique id assigned by the vendor to the device */

    uint64_t system_get_ticks(void);        /* internal number of processor tick */
    uint64_t system_get_time_nsec(void);    /* current time in nanosecond        */

    uint32_t self_get_id(void);          /* assigned id to this container on the device */
    uint32_t self_get_lifetime(void);    /* remaining container lifetime in second      */

    uint64_t sensors_get_temperature(void);          /* temperature in 1/1000th of kelvin   */
    uint64_t sensors_get_value(uint8_t sensor_id);   /* retrieve value of a board sensor    */

    uint64_t algorithms_get_random(void);    /* 64 bit random generator */
```

```
size_t io.write(int fd, void *buf, size_t n); /* write n bytes of buf to fd */
size_t io.read(int fd, void *buf, size_t n);  /* read n bytes into buf from fd */
```

Listing 2: Runtime Containers Service API

Obviously, how this API is exposed to containers depends on the interpreter and some adjustments may be required. Moreover, the API relies on a communications channel between the container and the RCS. This channel is sensible because it allows exchanges with the isolated container. Implementation of this channel is ongoing and we will detail it at a later stage of the project.

3.3 Federated Machine Learning Service API

The API will provide primitives which allow containers to manage or personalize federated learning computation.
It also covers computation of privacy budget analytically while training FL models by following procedure explained in 2.5. These trained model parameters on fulfilling privacy requirements are deployed using an inference C++ script to do prediction on unseen data points.
The Federated Machine Learning module's Client and Server routines are written in C++ in a modular fashion which allows flexibility in terms of learning algorithm, model architectures, etc. Communication between server and clients is established using RIOT's Network API using CoAP protocol where different REST like endpoints are exposed to read and write model parameters.

References

[1] The Pip development team , “Website of: The pip protokernel.” `http://pip.univ-lille1.fr/`, 2022. [Online; accessed January 12, 2022].

[2] ARM , “Website of: Armv8-m memory protection unit version 2.0.” `https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf/`, 2017. [Online; accessed March 09, 2021].

[3] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, “Deep learning with differential privacy,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, oct 2016.