Journal of
Low Power Electronics and Applications

Article

Efficient ROS-Compliant CPU-iGPU Communication on Embedded Platforms

Mirco De Marchi, Francesco Lumpp *, Enrico Martini, Michele Boldo, Stefano Aldegheri and Nicola Bombieri

Department of Computer Science, University of Verona, 37134 Verona, Italy; mirco.demarchi@studenti.univr.it (M.D.M.); enrico.martini_02@studenti.univr.it (E.M.); michele.boldo@studenti.univr.it (M.B.); stefano.aldegheri@univr.it (S.A.); nicola.bombieri@univr.it (N.B.)
* Correspondence: francesco.lumpp@univr.it

Citation: De Marchi, M.; Lumpp, F.; Martini, E.; Boldo, M.; Aldegheri, S.; Bombieri, N. Efficient ROS-Compliant CPU-iGPU Communication on Embedded Platforms. *J. Low Power Electron. Appl.* **2021**, *11*, 24. https://doi.org/10.3390/jlpea11020024

Academic Editors: Andrea Acquaviva and Francesco Barchi

Received: 6 May 2021
Accepted: 22 May 2021
Published: 26 May 2021

Abstract: Many modern programmable embedded devices contain CPUs and a GPU that share the same system memory on a single die. Such a unified memory architecture (UMA) allows programmers to implement different communication models between CPU and the integrated GPU (iGPU). Although the simpler model guarantees implicit synchronization at the cost of performance, the more advanced model allows, through the zero-copy paradigm, the explicit data copying between CPU and iGPU to be eliminated with the benefit of significantly improving performance and energy savings. On the other hand, the robot operating system (ROS) has become a de-facto reference standard for developing robotic applications. It allows for application re-use and the easy integration of software blocks in complex cyber-physical systems. Although ROS compliance is strongly required for SW portability and reuse, it can lead to performance loss and elude the benefits of the zero-copy communication. In this article we present efficient techniques to implement CPU–iGPU communication by guaranteeing compliance to the ROS standard. We show how key features of each communication model are maintained and the corresponding overhead involved by the ROS compliancy.

Keywords: CPU–iGPU communication; ROS; cyber-physical systems; CUDA

1. Introduction

Traditional CPU–iGPU communication models are based on data copy. The physically shared memory space is partitioned into logical spaces, and the CPU exchanges data with the iGPU by copying the data from its own partitions to the iGPU partitions and vice-versa. Both the CPU and iGPU caches can hide the data copy overhead, and cache coherence is guaranteed implicitly by the operating system that flushes the caches before and after each GPU kernel invocation [1].

To ease the CPU-GPU programming and avoid explicit data transfer invocations, user-friendly solutions have been proposed to allow programmers to implement CPU–iGPU communication through data pointers. In this communication model, the physically shared memory is still partitioned into CPU and GPU logic spaces, although they are abstracted and used by the programmer as a virtually unified logic space. The runtime system implements synchronization between CPU and iGPU logical spaces through an on-demand page migration heuristic [2]. To avoid physical data copy between CPU and GPU memories, communication based on zero-copy has been introduced to pass data through pointers to the *pinned* shared address space. When CPU and the integrated GPU (iGPU) physically share the memory space, their communication does not rely on the PCIe bus and zero-copy communication can be sensibly more efficient [3]. Zero-copy communication is instrumental for performance and energy efficiency in many real-time applications. Examples are camera- or sensor-based applications, in which the CPU offloads streams of data to the GPU for high-performance and high-efficiency processing [4].

Zero-copy allows for synchronous accesses of CPU and iGPU to the shared address space and, as a consequence, it requires the system to guarantee cache coherency across the memory hierarchy of the two processing elements. The overhead involved by SW coherency protocols applied to CPU–iGPU devices often eludes the benefit of the zero-copy communication. To avoid such an overhead, many systems solve the problem at the root by disabling the last level caches (LLC) [5].

On the one hand, zero-copy guarantees the best performance when adopted in applications in which CPU is the data producer and the GPU is the consumer (i.e., data-flow applications). On the other hand, it often leads to strong performance degradation when the applications make intensive use of the GPU cache (i.e., cache-dependent applications). In these cases, communication models based on data copy are often the best solution. As a trade-off solution, more recent embedded devices include hardware-implemented I/O coherence, by which the iGPU directly accesses the CPU cache, while the GPU cache is still disabled [6].

Even though some work has been proposed to study the best choice among the different communication models by considering the application characteristics [7], the effects of making these models compliant to the standard communication protocols adopted for the development of cyber-physical systems (CPS) has never been investigated. The design of a CPS, e.g., a robotic system, involves experts from different application domains [8–10], which must integrate the pieces of software developed to address the specific issues from each distinct domain. However, such an integrated design is still an open problem [11–14]. In recent years, to facilitate the integration of such multi-domain components and (sub)systems, distributed frameworks, e.g., the robot operating system (ROS) [15] has been proposed to provide the necessary support to develop the distributed software for system components and devices. ROS has been proposed as a flexible framework for developing robotic and cyber-physical systems software. It is a collection of APIs, libraries, and conventions that aim at simplifying the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It has become a de-facto reference standard for developing robotic and cyber-physical system applications. Compliance to ROS is nowadays a key aspect for the application, re-use, and easy integration of software blocks in complex systems. Although ROS compliance is nowadays strongly required for developing cyber-physical systems and in particular robotic systems, it can lead to performance loss and sensibly impact on the CPU–iGPU communication model characteristics.

In this article we present different techniques to efficiently implement CPU–iGPU communication on UMA that complies with the ROS standard. We show that a direct porting of the most widespread communication protocols to the standard ROS can lead to up to 5000% overhead, often eluding the benefits of the GPU acceleration. This article addresses this issue by presenting several dedicated solutions to implement such ROS-compliant communication protocols that (i) take advantage of the CPU–iGPU communication architecture provided by the embedded device, (ii) apply different mechanisms included in the ROS standard, and (iii) are customized each one to a different communication scenario, from two to many ROS nodes.

The article presents the results obtained by applying the proposed models to implement communication among different ROS nodes and on different architecture (i.e., NVIDIA Jetson TX2, Jetson Xavier with I/O coherence) and by considering different applications (i.e., data-flows, and cache-dependent applications). The result show that, by selecting the right model, the overhead introduced by making the CPU–iGPU communication compliant to ROS is negligible.

2. Related Work

Some previous work on cyber-physical system has seen the integration of ROS, cyber-physical systems, and embedded boards. In [16], the authors used ROS combined with Gazebo to create a 3D visualization of a fixed-multi wing unmanned aerial vehicle. In [17], the authors present the integration of intelligent manufacturing systems, which may required smart re-configuration. It has been developed on top of ROS Jade (ROS 1) to deploy software on autonomous transport vehicles. In [18], the authors present Autoware software, which is designed to enable autonomous vehicles on embedded boards, specifically Nvidia Drive PX2 boards communicating through ROS Kinetic (ROS 1), showing high levels of efficiency thanks to the iGPU and ARM CPU installed on the embedded board, superior to high-end x86 laptops parts.

Unlike these prior works, that target the cyber-physical systems from the point of view of functionality or performance, the analysis and methodology proposed in this work target the overhead of CPU–iGPU communication through the ROS protocols on *zero-copy* communication models that, to the best of our knowledge, has not been addressed in prior works.

3. CPU-iGPU Communication and ROS Protocols

This section presents an overview of the CPU–iGPU communication models and the standard ROS-based communication between tasks. For the sake of clarity and without loss of generality, we consider CUDA as the iGPU architecture.

3.1. CPU–iGPU Communication in Shared-Memory Architectures

The most traditional and simple communication model between CPU and iGPU is the *Standard Copy* (*CUDA-SC* in the following), which is based on explicit data copy (see Figure 1a). With CUDA-SC, the physically shared memory space is partitioned into different logical spaces and the CPU copies the data from its own partitions to the iGPU partitions. Since both the CPU and GPU caches are enabled, it can hide the data copy overhead. The cache coherence is guaranteed implicitly by the operating system, which flushes the caches before and after each GPU kernel invocation. Such a model also guarantees implicit synchronization for data access between CPU and iGPU.

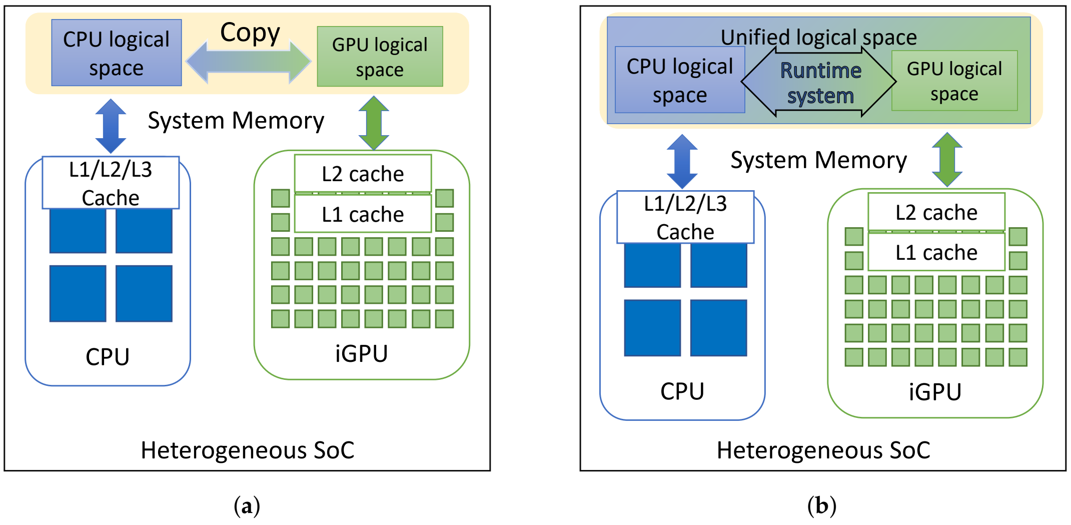

Figure 1. Copy-based CPU–iGPU communication methods. (**a**) Communication on an UMA heterogeneous SoC using CUDA-SC; (**b**) Communication on an UMA heterogeneous SoC using CUDA-UM.

Programmers can take advantage of different solutions that allow for this explicit copy to be removed and for performance and power consumption benefits.

The *unified memory* model (*CUDA-UM* in the following) allows the programmer to implement CPU–iGPU communication through data pointers, thus avoiding explicit data transfer invocations (see Figure 1b). In this communication model, the physically shared memory is still partitioned into CPU and iGPU logic spaces although they are abstracted and used by the programmer as a virtually unified logical space. The runtime system implements synchronization between CPU and iGPU logical spaces through an *on-demand page migration* heuristic.

The zero-copy model (*CUDA-ZC* in the following) implements CPU–iGPU communication by passing data through pointers to the *pinned* shared address space. When CPU and the iGPU physically share the same physical memory space, their communication does not rely on the PCIe bus and communication through CUDA-ZC provides the best efficiency in different application contexts [3,4]. CUDA-ZC through shared address space requires the system to guarantee cache coherency across the memory hierarchy of the two processing elements. Since the overhead involved by SW coherency protocols applied to CPU–iGPU devices may elude the benefit of the zero-copy communication, many systems address the problem by disabling the last level caches (see Figure 2a).

CUDA-ZC best applies when the shared address space between CPU and iGPU points to the same physical memory space but, if the memory data pointer changes during CPU execution due to new data allocations, CUDA-ZC requires a new allocation. However, the CUDA-ZC model provides the `cudaHostRegister` function API to handle this situation without allocating additional memory. It applies *page-locking* to a CPU memory address and transforms it into a CUDA-ZC address. This operation requires hardware I/O coherency to avoid memory consistency problems caused by CPU caching.

Although additional memory has to be allocated for coherency with `cudaHostAlloc` API, the data copy phase is more efficient than CUDA-SC as it avoids the copy stream buffering. On the other hand, since a copy is performed, this situation decreases the performance compared to the zero-copy model if the copy is performed on the same memory location.

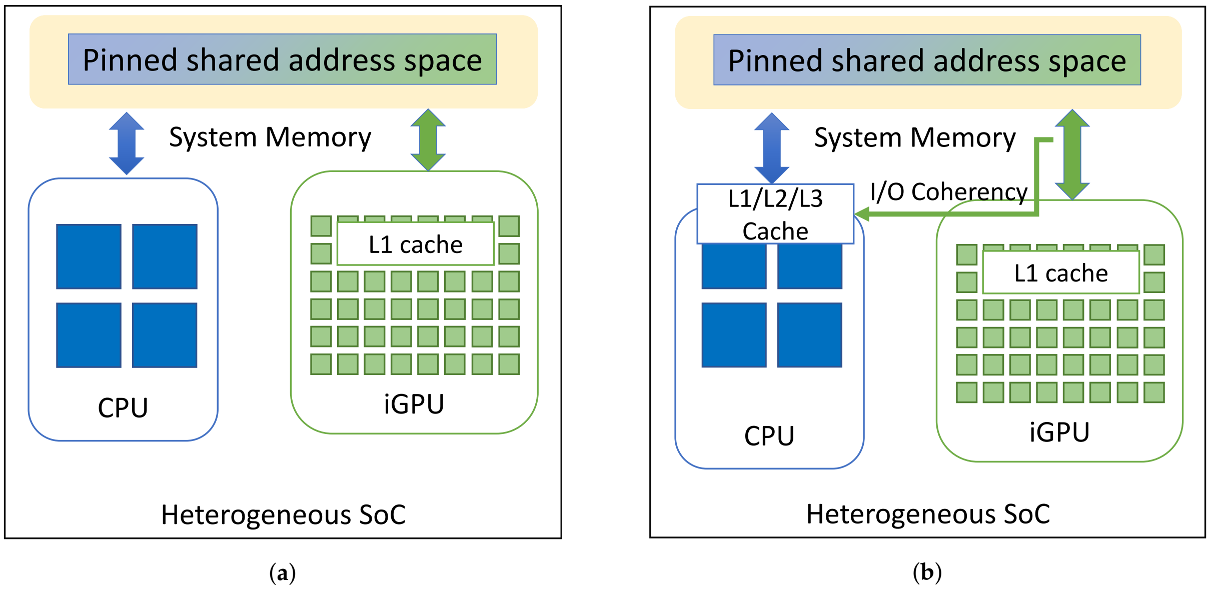

Figure 2. Zero-copy CPU–iGPU communication methods. (**a**) Communication on an UMA heterogeneous SoC using CUDA-ZC; (**b**) Communication on an UMA heterogeneous SoC using CUDA-ZC and I/O coherency.

Although CUDA-ZC best applies to many SW applications (e.g., applications in which CPU is the data producer and the GPU is the consumer), it often leads to strong performance degradation when the applications make intensive use of the GPU cache (i.e., cache-dependent applications). To reduce such a limitation, more recent embedded devices include hardware-implemented I/O coherence, which allows the iGPU to snoop the CPU cache. Indeed, the CPU cache is always active and any update on it is visible to the GPU. In contrast, the GPU cache is disabled (see Figure 2b).

3.2. ROS-Based Communication between Nodes

In ROS, the system functionality is implemented through communicating and interacting nodes. The nodes exchange data using two mechanisms: the *publisher-subscriber* approach and the *service* approach (i.e., remote procedure call—RPC) [19].

The publisher–subscriber approach relies on *topics*, which are communication buses identified by a name [20]. One node (publisher) publishes data asynchronously and one node (subscriber) receives the data simultaneously. Given a publisher URI, a subscribing node negotiates a connection through the master node via XML RPC. The result of the negotiation is that the two nodes are connected, with messages streaming from publisher to subscriber using the appropriate transport.

Each transport has its own protocol for how the message data is exchanged. For example, using TCP, the negotiation would involve the publisher giving the subscriber the IP address and port on which to connect. The subscriber then creates a TCP/IP socket to the specified address and port. The nodes exchange a connection header that includes information like the MD5 sum of the message type and the name of the topic, and then the publisher begins sending serialized message data directly over the socket.

The topic-based approach relies on a socket-based communication, which allows for collective communication. The communication channel is instantiated at the system start-up and never closed. In this type of communication there can be many publishers and many subscribers on the same topic, as in the example of Figure 3.

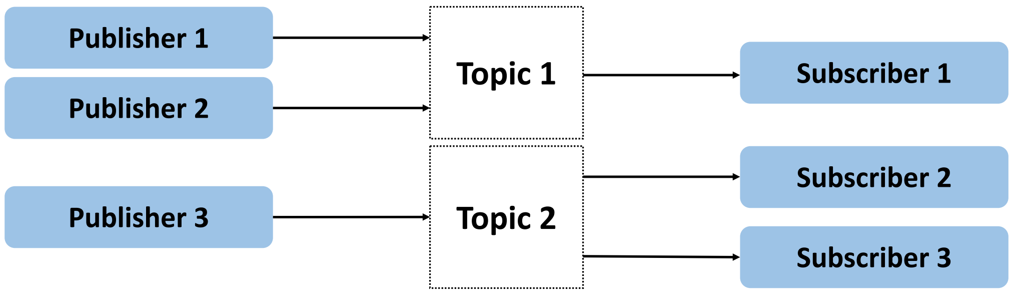

Figure 3. ROS publisher–subscriber communication approach.

With the service approach, a node provides a service on request (see Figure 4). Any client can query the service synchronously or asynchronously. In case of synchronous communication, the querying node waits for a response from the service. There can be multiple clients, but only one service server is allowed for a given service.

In general, the service approach relies on a point-to-point communication (i.e., socket-based) between client and server and, for each client request, the server creates a dedicated new thread to serve the response. After the response, the communication channel is closed.

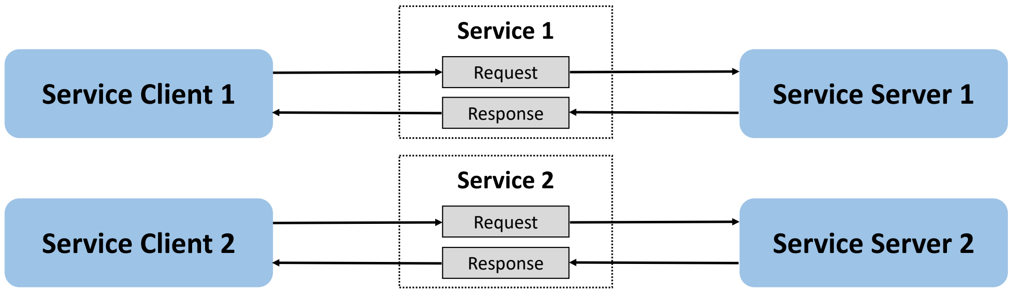

Figure 4. ROS service communication approach.

Publishers–subscribers communication is implemented through a physical copy of the data. In case of nodes running on the same device and sharing the same resources, the copy of large-size messages may slow down the entire computation. ROS2 introduced *ROS zero-copy*, a new method to perform efficient intra-process communication [21].

Figure 5 shows the overview of such a communication method. With ROS zero-copy (*ROS-ZC* in the following), two nodes exchange the pointer to the data through topics, while the data are shared (not copied) in the common physical space. Even though such a zero-copy model guarantees high communication bandwidth between nodes instantiated on the same process, it has several limitations that prevent its applicability. First, it does not apply to service-based communication and it does not apply to inter process communication. In addition, it does not support multiple concurrent subscribers and it does not allow for computation-communication overlapping.

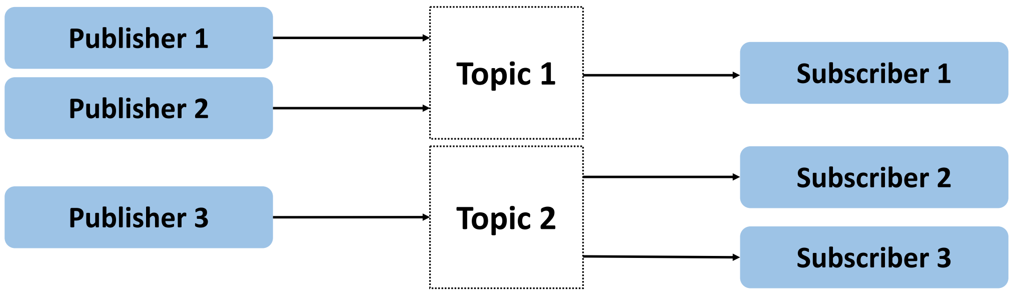

Figure 5. ROS2 intra-process communication.

4. Efficient ROS-Compliant CPU–iGPU Communication

GPU-accelerated code (i.e., GPU kernels) cannot be directly invoked by a ROS-compliant node. To guarantee ROS standard compliance of any task accelerated on GPU, the GPU kernel has to be wrapped to become a ROS node.

4.1. Making CPU–iGPU Communication Compliant to ROS

We consider, as a starting point, communication between CPU and iGPU implemented through CUDA-SC or CUDA-UM (see Figure 1a,b). For the sake of space and without loss of generality, we consider the performance of UM similar to SC. As confirmed by our experimental results, the maximum difference between the two model performance is within ±8% in all the considered devices. The difference is strictly related to the driver implementation for the on-demand page migration. Compared to the difference between CUDA-SC(CUDA-UM) and CUDA-ZC, we consider negligible, in this article, the difference of performance between CUDA-SC and CUDA-UM.

Figure 6 shows the most intuitive solution to make such a communication model compliant to ROS. The main CPU task and the wrapped GPU task are implemented by two different ROS nodes (i.e., processes P1 and P2). Communication between the two nodes relies on the publisher-subscriber model. The CPU node (i.e., process P1) has the role of executing the CPU tasks and managing the synchronization points between the CPU and GPU nodes. The CPU node publishes input data into a send topic and subscribes on a receive topic to get the data elaborated by the GPU. The GPU node (i.e., process P2) is implemented by a GPU wrapper and the GPU kernel. The GPU wrapper is a CPU task that subscribes on the send topic to receive the input data from P1, it exchanges the data with the GPU through the standard CPU–iGPU communication model, and publishes the result on the receive topic.

This solution is simple and easy to implement. On the other hand, the system keeps three synchronized copies of the I/O data messages (i.e., for CPU task, GPU wrapper, and GPU task). For each new input data received by the wrapper through the send topic, the memory slot allocated for the message has to be updated both in the CPU logic space (i.e., for the GPU wrapper) and in the GPU logic space (i.e., for the GPU task) as for the CUDA-SC standard model. On the other hand, CPU and GPU tasks can overlap, as the publisher–subscriber protocol allows for asynchronous and non-blocking communication.

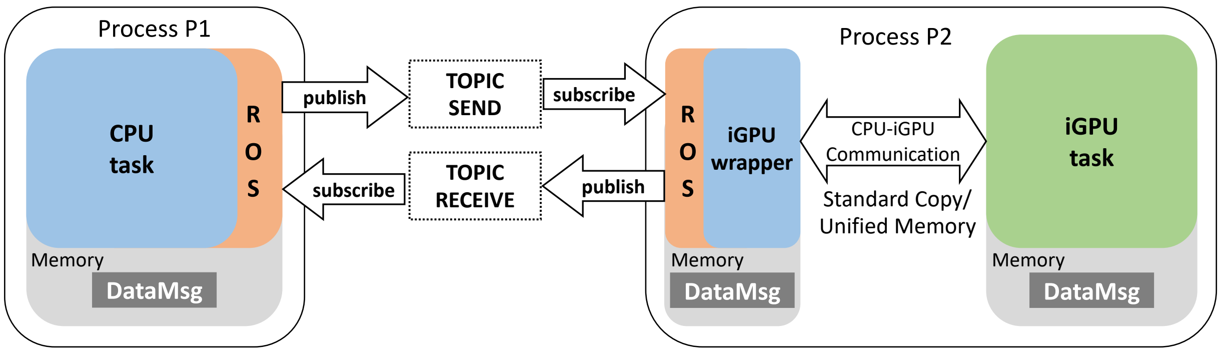

Figure 6. CPU-iGPU standard copy with ROS topic paradigm.

The following snippet of pseudo-code shows the steps performed by the CPU node to perform the CPU operations and to communicate with the iGPU node through the publish–subscribe paradigm and topics:

```
callback(msg::SharedPtr msg) {
copy_gpu_result(this->data, msg->data);

// synchronize operations and perform next step
sync_and_step(msg, this->data);
}

// init
this->publisher = create_pub<msg>(TOPIC_SEND);
this->subscriber = create_sub<msg>(TOPIC_RECEIVE, &callback);
// perform first step
msg::SharedPtr msg = new msg();
msg->data = this->data;
this->publisher->publish(msg); //< send to iGPU
cpu_operations(this->data); //< perform CPU ops in overlapping
```

Figure 7 shows a ROS-compliant implementation of the CUDA-SC model through the ROS service approach. Similarly to the first publisher–subscriber solution, the memory for the data message is replicated in each ROS node and in the GPU memory. The message has to be copied during every data communication and, since the service request can be performed asynchronously, CPU–iGPU operations can be performed in parallel.

The following snippet of pseudo-code shows the steps performed by the CPU node in order to perform the CPU operations, sends a request and waits for a response through RPC model:

```
// init
auto gpu_client = create_client<srv>(SERVICE_GPU_NAME);
while (!gpu_client->wait_for_service()); //< wait for GPU service available

// send a new request asynchronously
auto request = new request();
request->data = this->data;
auto gpu_result = gpu_client->async_send_request(request);

cpu_operations(this->data); //< perform CPU ops in overlapping

// wait for result
spin_until_future_complete(gpu_result);
copy_gpu_result(this->data, gpu_result.get()->data);
```

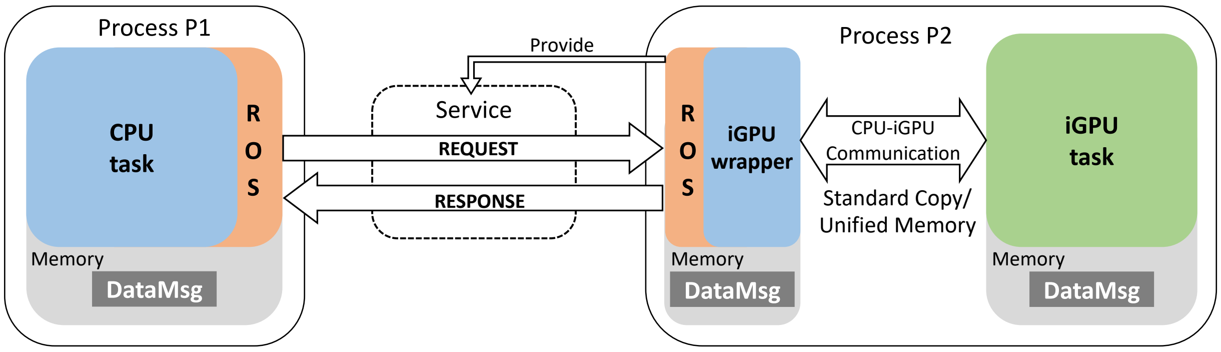

Figure 7. CPU-iGPU standard copy with ROS wrapper service paradigm (RPC).

Figure 8 shows a more optimized version of such a standardization, which relies on the ROS-ZC and intra-process communication. The ROS nodes are implemented as threads of the same process. As a consequence, each node shares the same virtual space memory and communication can rely on the more efficient protocols based on shared memory. Nevertheless, the usage of ROS-ZC has several limitations:

- ROS-ZC can be implemented only for communication between threads of the same process. When a zero-copy message is sent to a ROS node of a different process (i.e., inter-node communication), the communication mechanism automatically switches to the ROS standard copy;
- ROS-ZC does not allow for multiple nodes subscribed on the same data resource. If several nodes have to access a ROS-ZC message concurrently, ROS-ZC applies to only one of these nodes. The communication mechanism automatically switches to the ROS standard copy for the others. This condition holds for both intra-process and inter-process communication;
- ROS-ZC only allows for synchronous ownership of the memory address. A node that publishes a zero-copy message over a topic will not be allowed to access the message memory address. For this reason, CPU and iGPU operations cannot be performed in parallel, as the CPU node cannot execute operations after sending the message. Then the CPU node is forced to compute its operations either before or after the GPU operations (i.e., no overlapping computation is allowed over the shared memory address).

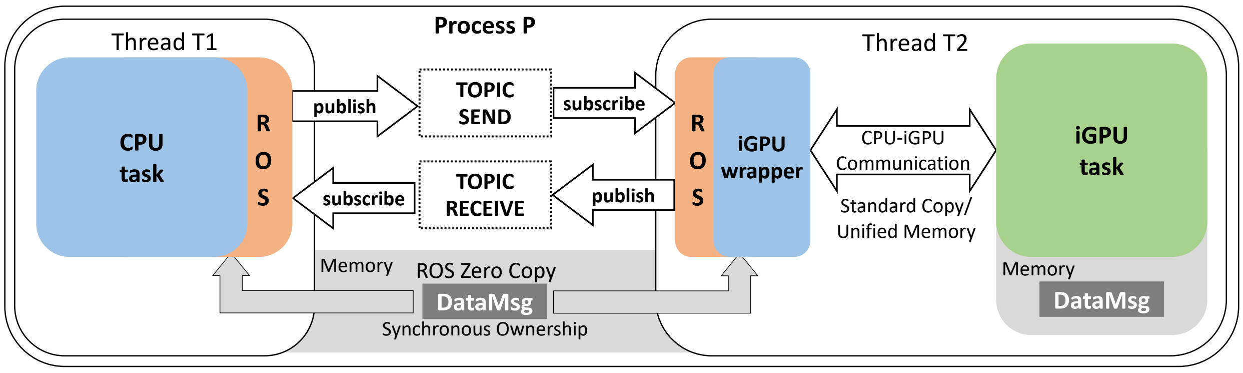

Figure 8. CPU–iGPU standard copy with ROS native zero-copy and topic paradigm.

These limitations guarantee no race conditions in shared memory locations and provide better performance with regards to ROS standard copy in case of intra-process communication. On the other hand, it does not apply for inter-process communication, which is fundamental for portability of robotic applications.

The code for ROS-ZC slightly changes compared to the ROS-SC topic paradigm. The message is defined as `UniquePtr` rather than `SharedPtr` and all the operations on the data have to be performed before the message is published.

```
callback(msg::UniquePtr msg) {
// ...
cpu_operations(msg->data); //< perform CPU ops before publish
this->publisher->publish(std::move(msg)); //< send to iGPU
// here I can't do anything on msg->data
}

// init
msg::UniquePtr msg = new msg();
// ...
```

Due to the several limitations involved by the previous simple solutions, we propose a new approach that maintains the standard ROS interface and related modularity advantages while taking advantage of the shared memory between processes when nodes are deployed in the same unified memory architecture (see Figure 9). The idea is to implement a ROS interface that shares the reference to the inter-process shared memory with the other ROS nodes, and exchanges only synchronization messages between nodes. The proposed solution has the following characteristics:

- Not only intra-process : This solution also applies to inter-process communication by means of a IPC shared memory managed by the operating system;
- Unique data allocation: the only memory allocated for data exchange is the shared memory;
- Efficient CUDA-ZC: The reference to the shared memory does not change during the whole communication process between ROS nodes. As a consequence, it also applies to the CUDA-ZC communication between the wrapper and the iGPU task. (see Figure 10);
- Easy concurrency: The shared memory can be accessed concurrently by different nodes allowing parallel execution between nodes over the same memory space when the application is safe from race condition.

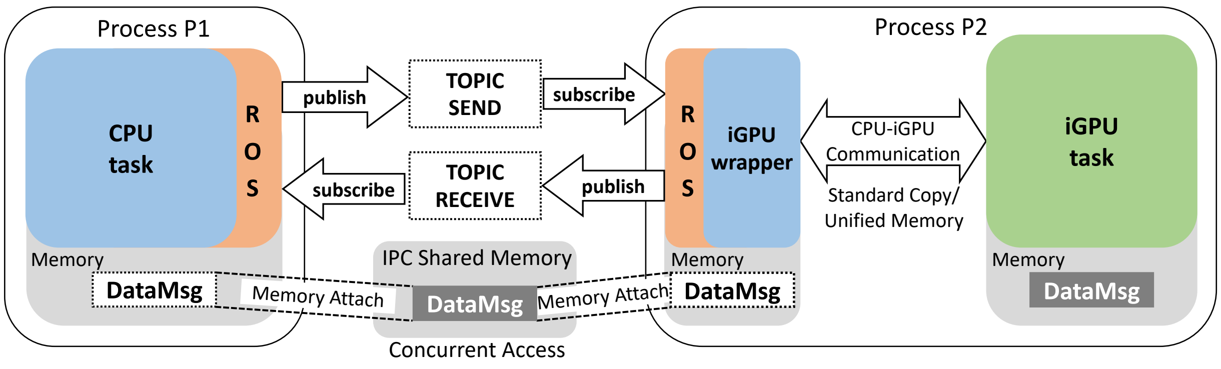

Figure 9. CPU–iGPU standard copy with the proposed ROS-ZC solution and topic paradigm.

The two drawbacks of this implementation are the risk of race conditions, which have to be managed by the programmer, and the need to fall back to the standard ROS communication protocol when one of the nodes moves outside of the unified memory architecture.

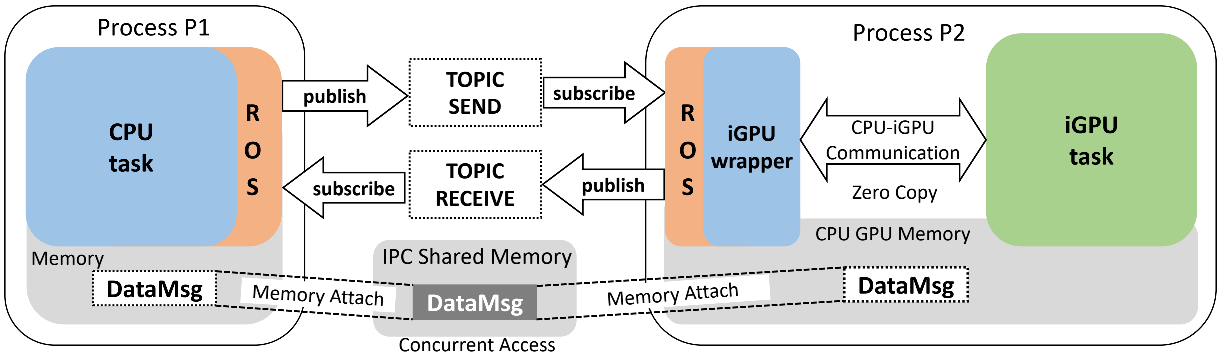

Figure 10. CPU–GPU zero-copy with our ROS zero-copy solution and topic paradigm.

The following pseudo-code snippet shows IPC shared management and sync message communication in the CPU node:

```
callback(sync_msg::SharedPtr msg) {
// no copy needed
if (msg->status == SyncStatus::DONE) {
// synchronize operations and perform next step
sync_and_step(msg, this->data);
}
}

// init
auto this->publisher = create_pub<sync_msg>(TOPIC_SEND);
auto this->subscriber = create_sub<sync_msg>(TOPIC_RECEIVE, &callback);

// IPC Shared Memory creation
int shmid = shmget(SHM_KEY, size, 0666 | IPC_CREAT);
if (shmid == -1) error();

// IPC Shared Memory attach
this->shm_msg_ptr = (msg *) shmat(shmid);

// start synchronize GPU node
sync_msg::SharedPtr msg = new sync_msg();
msg->status = SyncStatus::START;
msg->shm_key = SHM_KEY;
this->publisher->publish(msg);

cpu_operations(this->shm_msg_ptr); //< perform CPU ops in overlapping
```

4.2. Making Multi-Node CPU–iGPU Communication Compliant to ROS

The most intuitive ROS integration in a concurrent GPU–iCPU application is the partition of the CPU and iGPU tasks into two different nodes. In case of multiple nodes (CPUs and iGPU), we propose a different architecture by which a dedicated node implements exclusively the multiple node synchronization and scheduling (see Figure 11). In particular, the CPU nodes wait for any new data message from the send topic, perform the defined tasks, and then return the results in the corresponding topic. The iGPU node(s) performs similarly for the CUDA kernel tasks. Although the scheduler node acts as a synchronizer for the CPU and iGPU nodes, it provides the data to elaborate on send topic, waits for CPUs and iGPU responses and, when both are received, it merges the responses and forwards the merged data.

Figure 11 shows the multi-node architecture implemented as an extension of the two node architecture based on the *ROS topic paradigm*. It can be analogously implemented with *ROS service paradigm*.

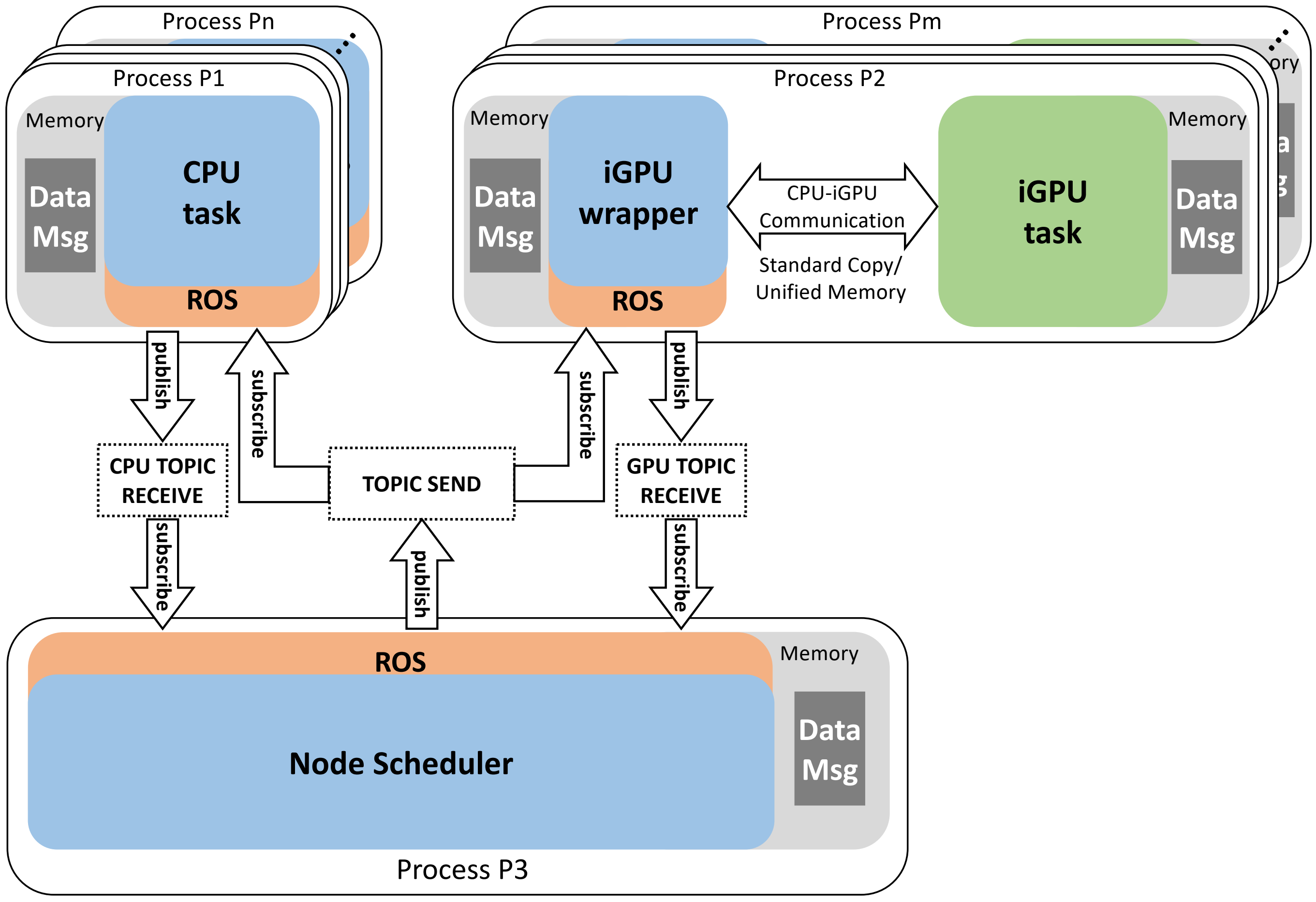

Figure 11. CPU–GPU Standard Copy with ROS topic paradigm in multi-node architecture.

Comparing the two solutions with multi-node architecture, the topic paradigm is more efficient than the service paradigm in terms of communication as the sending topic is shared between the other subscriber nodes. In the service paradigm, a new request has to be performed for each required service. This aspect is particularly important as it underlines the scalability advantages of the multi-node architecture compared to the two node architecture. Assuming an application with N CPU tasks and M GPU tasks all sharing the same resource. The system can be implemented compliant to ROS with one scheduler node, N CPU nodes, and M GPU nodes. All nodes are synchronized by the scheduler node with the ROS topic or ROS service paradigm. Considering the topic paradigm, the system requires $N + M$ receive topics and one send topic. With the service paradigm, each $N + M$ CPU and GPU nodes have to create a service, which will be used by the scheduler node.

The topic paradigm relies on an single send topic, which can be shared by different subscribers. With the service paradigm, the scheduler node has to perform a new request for each service. In terms of performance, the topic paradigm for multiple tasks is more efficient in the case of data-flow applications.

In order to implement multi-node architecture, it is necessary to create N callbacks for CPU data receive and M callbacks for GPU data receive. In this case, the scheduling node will have the only purpose to manage the communication between the others node. The scheduling implementation skeleton for the topic paradigm is the following:

```
callback_cpu_i(msg::SharedPtr msg) {
copy_cpu_i_result(this->data, msg->data);
// synchronize operations and perform next step
sync_and_step(msg, this->data);
}

callback_gpu_j(msg::SharedPtr msg) {
copy_gpu_j_result(this->data, msg->data);
// synchronize operations and perform next step
sync_and_step(msg, this->data);
}

// init
this->publisher = create_pub<msg>(TOPIC_SEND);
this->subscriber_cpu_i = create_sub<msg>(TOPIC_CPU_I_RECEIVE,
&callback_cpu_i);
this->subscriber_gpu_i = create_sub<msg>(TOPIC_GPU_J_RECEIVE,
&callback_gpu_j);
// perform first step
msg::SharedPtr msg = new msg();
msg->data = this->data;
this->publisher->publish(msg); //< send to subscriber nodes
```

Figure 12 shows the multi-node architecture implemented through the ROS-ZC for node communication. The CPU node, the iGPU node, and the scheduler node have the same roles as before, but they are executed as threads of the same process and the exchanged messages rely on the zero-copy paradigm. For this reason, two communicating nodes (i.e., scheduler-CPU or scheduler-iGPU) can exchange only the reference address of the data. The communication concerning the remaining competing node automatically switches to the ROS standard copy modality. For example, scheduler-CPU communication will be switched to ROS standard copy if scheduler-iGPU transfer is ROS-ZC and scheduler-iGPU will be ROS-SC if scheduler-iGPU is ROS-ZC. The zero-copy exchange will be provided to the first node ready to receive the data and the choice of such a node is not predictable. In general, this architecture with N CPU tasks and M iGPU tasks requires $N + M - 1$ data instances in the virtual address memory space (e.g., for one CPU task and one iGPU the system requires two data instances, as showed in Figure 12), as only one instance can be shared in zero-copy between the scheduler node and another node. Therefore, due to the ROS-ZC limitations, this architecture with ROS-ZC can save only one data instance compared to the ROS-SC solution. The advantage of the multi-node architecture with ROS-ZC is that it overcomes the limitations of the two node architecture with ROS-ZC, which does not allow for concurrent CPU and iGPU execution. Thanks to the scheduler node, which implements CPU-iGPU synchronization, all CPU and iGPU tasks can be executed while overlapping.

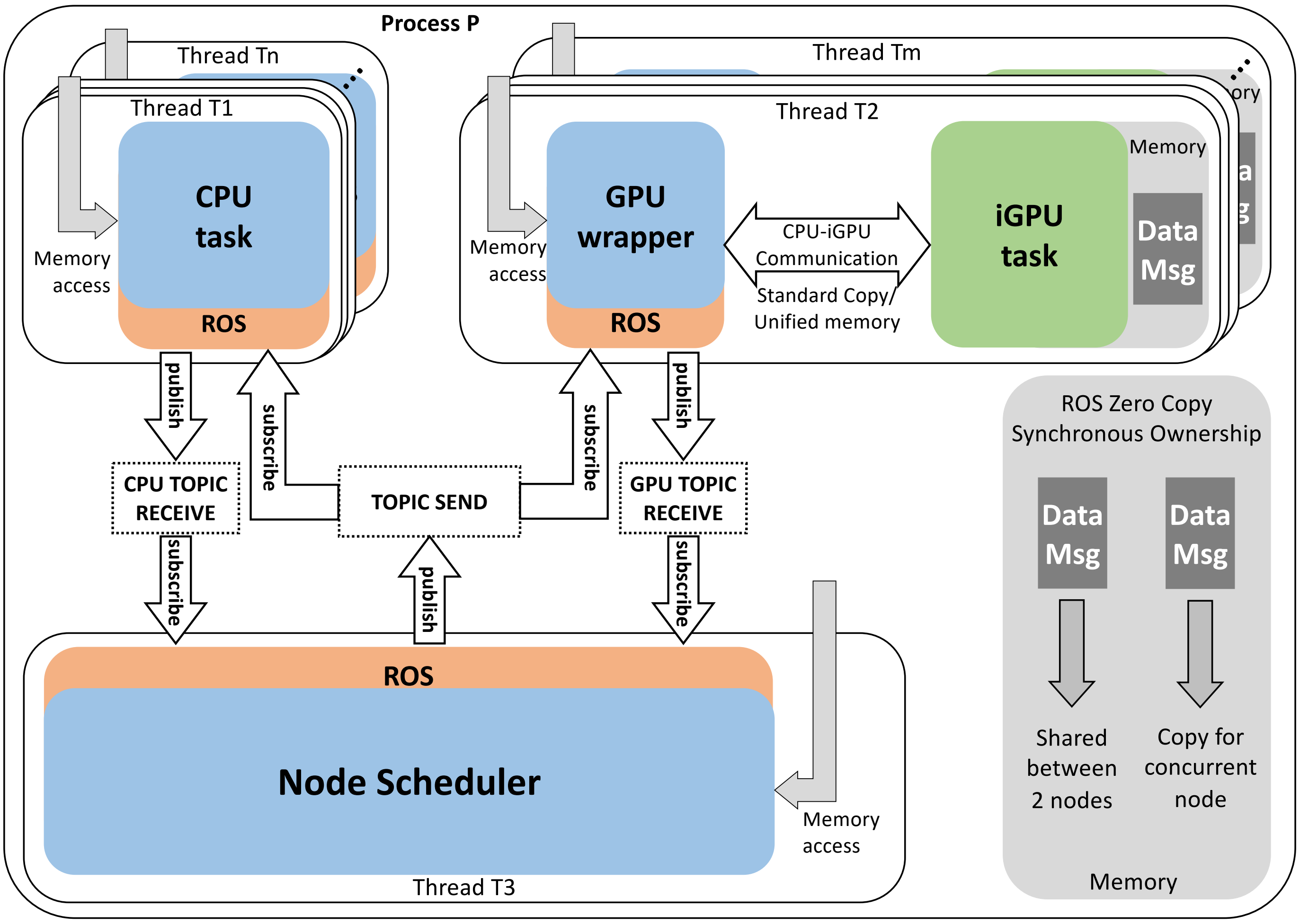

Figure 12. CPU–GPU SC with ROS native zero-copy, topic paradigm, and multi-node architecture.

However, since data copy is still needed, it is not possible to combine ROS-ZC to CUDA-ZC in case of multiple nodes by fully taking advantage of the zero-copy semantics. This is due to the fact that ROS does not guarantee that the same virtual address is maintained for the GPU node. As a consequence, as confirmed by our experimental results, this solution cannot guarantee the best performance in multi-node communication compared to standard copy solutions in terms of both memory usage and GPU management.

To overcome such a limitation, we propose the solution represented in Figure 13. Differently from the previous solution, the system manages the IPC shared memory and, unlike ROS-ZC, the nodes can be instantiated as processes. The scheduler node creates the IPC shared memory by using the standard Linux OS syscall, Then, it performs the memory attach to bind the shared created memory to its own virtual memory space, and, then, it shares with the other nodes synchronization messages with the shared memory information. The CPU nodes and iGPU nodes wait for synchronization messages. They obtain the IPC shared memory with the OS syscalls, they perform the memory attach, and they finally perform the requested actions. This approach relies on the ROS-ZC mechanism implemented through the shared memory. It applies for both the intra-process and inter-process communication. On the other hand, since it aims at avoiding multiple copies of the data message, it involves more overhead to manage race condition among the multiple nodes sharing the same logical space.

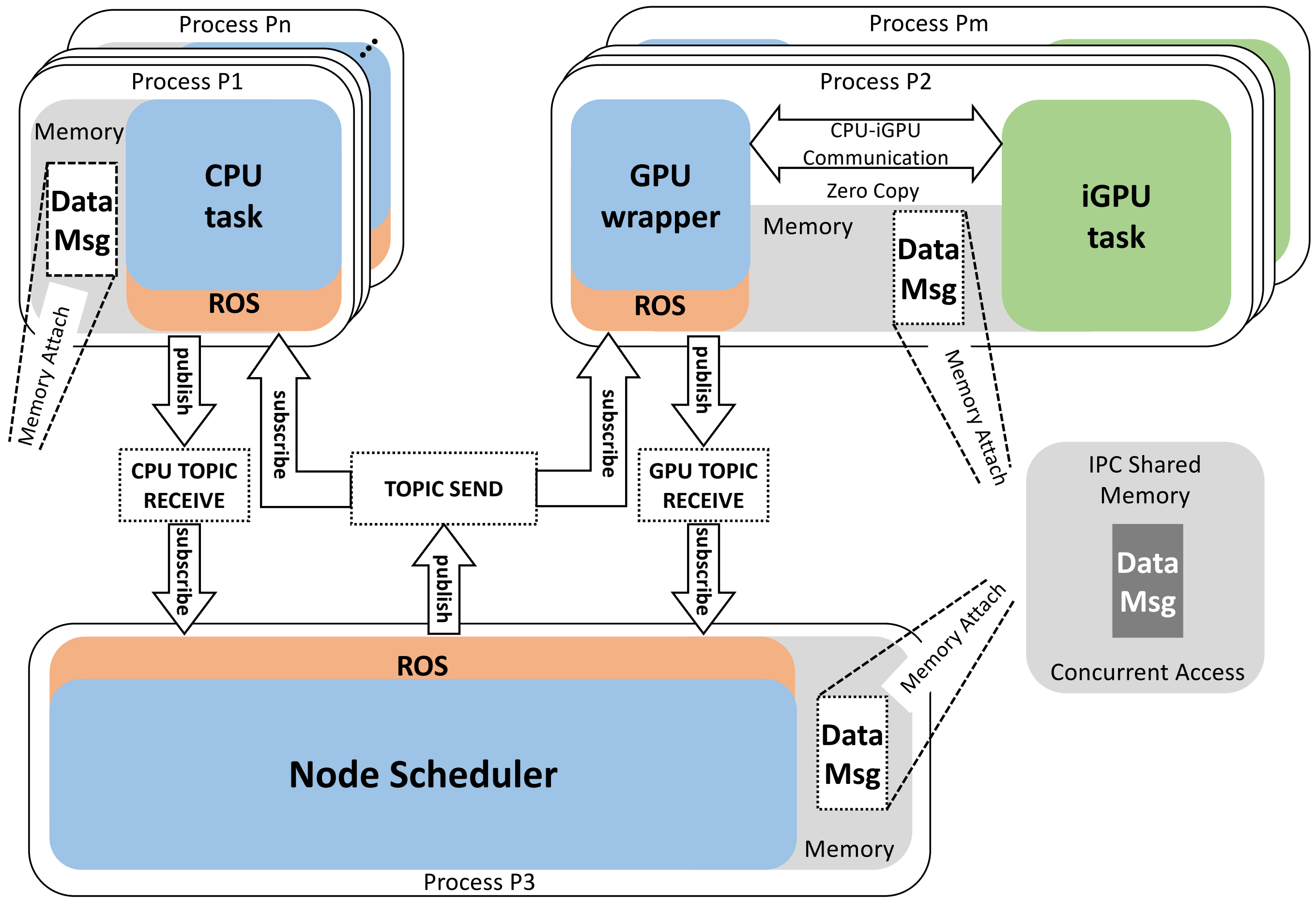

Figure 13. CPU–GPU zero-copy with our ROS zero-copy solution, topic and 3 nodes architecture.

5. Experimental Results

To verify the performance of the proposed ROS-compliant communication models, we carefully tailored two different benchmarks: cache-dependent and concurrent benchmark.

The cache benchmark, implements the elaboration of a matrix data structure independently performed by both CPU and iGPU. In particular, the CPU performs a series of floating point square roots, divisions, and multiplications with data read and written from and to a single memory address. The GPU performs a 2D reduction multiple times through linear memory accesses. This is achieved through iterative loading of the operands (`ld.global` instructions), a sum (`add.s32`), and the result store (`st.global`). As a consequence, this benchmark makes intensive use of the caches (both CPU and GPU), with no overlapping between CPU and iGPU execution. CUDA-ZC makes use of the concurrent execution of the routines and the concurrent access to the shared data structure. CUDA-SC explicitly exchanges the data structure before the routine computation.

The concurrent benchmark, performs a balanced CPU+GPU computation through a routine with highly reduced use of the GPU cache. The GPU kernel implements a single read access (`ld.global`) and single write access (`st.global`) per iteration in order to reduce the cache usage as much as possible. It implements a concurrent access pattern and a perfect overlap of the CPU and GPU computations to extrapolate the maximum communication performance the given embedded platform can provide by considering CUDA-ZC. It should greatly favor the communication patterns that allow concurrent access to the shared data (i.e., only CUDA-ZC).

The routines of both benchmarks are optimized on I/O coherent hardware through the use of the `cudaHostRegister` API.

For all tests, we used an NVIDIA Jetson TX2 and a Jetson Xavier as embedded computing architectures, which are prevalent low-power heterogeneous devices used in industrial robots, machine vision cameras, and portable medical equipment, see Figure 14. Both of these boards are equipped with iGPU and UMA, which allow us to best apply the proposed methodology. The TX2 consists of a quad-core ARM Cortex-A57 MPCore, a dual-core NVIDIA Denver 2 64-Bit CPU, 8 GB of unified memory and a NVIDIA Pascal iGPU with 256 CUDA cores. The Xavier consists of four dual-core NVIDIA Carmel ARMv8.2 CPUs, 32 GB of unified memory and a 512 CUDA cores NVIDIA Volta iGPU with I/O coherency.

Figure 14. The embedded platforms used for testing. The Nvidia Jetson Xavier (**left**) and Nvidia Jetson TX2 (**right**).

These two synthetic tests aim at maximizing the communication bottleneck and are representative of a worst-case scenario communication-wise. Real world applications, especially in the field of machine learning, will see a lesser bottleneck because of a more coarse-grained communication. In contrast, a high number of nodes with limited communication will still see benefits from the application of the proposed methodology to reduce the overall communication overhead.

Table 1 shows the performance results obtained by running the two benchmarks on the two different devices with different communication models (i.e., CUDA-SC, CUDA-ZC) without ROS. The reported times are the averaged results of 30 runs. Standard deviation is also considered in the fifth column (i.e., column "SD"). As expected, the TX2 device provides less performance then the Xavier. In both devices, the cache benchmark with the CUDA-ZC model, which disables the last level caches of CPU and GPU, provides the worst performance. The concurrent benchmark, even with a light cache workload and concurrent execution, still leads to performance loss. On the other hand, the I/O coherency implemented in hardware in the Xavier reduces this performance loss. In contrast, such a performance loss is extremely evident in the TX2.

The concurrent benchmark shows how a lighter cache usage combined to I/O coherency and concurrent executions thanks to CUDA-ZC, allows for significant performance improvements when compared to CUDA-SC.

The results of Table 1 will be used as reference times to calculate the *overhead* of all the tested ROS-based configurations.

Table 1. Time reference of cache benchmark and concurrent benchmark in NVIDIA Jetson TX2 and Xavier with CUDA-SC and CUDA-ZC.

NVIDIA Device	Benchmark	GPU Copy	Time (ms)		SD (ms)
Jetson TX2	Cache	CUDA-SC	833.0	±	3.8
Jetson TX2	Cache	CUDA-ZC	8509.1	±	131.7
Jetson TX2	Concurrent	CUDA-SC	1053.0	±	8.2
Jetson TX2	Concurrent	CUDA-ZC	1316.1	±	33.9
Jetson Xavier	Cache	CUDA-SC	207.7	±	9.8
Jetson Xavier	Cache	CUDA-ZC	244.7	±	11.0
Jetson Xavier	Concurrent	CUDA-SC	381.2	±	8.3
Jetson Xavier	Concurrent	CUDA-ZC	256.9	±	6.1

We present the results obtained with the proposed ROS compliant models into four tables. Tables 2 and 3 present the results on the Jetson TX2 with the cache and concurrent benchmarks, respectively. Tables 4 and 5 present the results on the Jetson Xavier with the cache and concurrent benchmarks, respectively. In the tables, the first column indicates the proposed ROS-compliant communication model, with a reference to the corresponding figure in the methodology section. The second column indicates the CPU–iGPU CUDA communication type. The third and forth columns indicate the ROS communication protocol. The fifth and sixth show the average run time of 30 executions and the standard deviation, respectively. The last column shows the overhead.

Table 2. Results for the Cache Benchmark on the NVIDIA Jetson TX2. The reference execution time for CUDA-SC is 833.0 ms and 8509.1 ms for CUDA-ZC.

Arch.	GPU Copy	ROS Type	ROS Copy	Time (ms)		SD (ms)	Overhead
2 nodes (Figure 6)	CUDA-SC	Topic	ROS-SC	22,825.2	±	5763.5	2640%
2 nodes (Figure 7)	CUDA-SC	Service	ROS-SC	22,715.0	±	3929.8	2627%
2 nodes (Figure 8)	CUDA-SC	Topic	ROS-ZC	1056.0	±	32.2	27%
2 nodes (Figure 8)	CUDA-ZC	Topic	ROS-ZC	10,254.2	±	189.0	21%
2 nodes (Figure 9)	CUDA-SC	Topic	ROS-SHM-ZC	852.6	±	7.7	2%
2 nodes (Figure 10)	CUDA-ZC	Topic	ROS-SHM-ZC	9474.0	±	226.3	11%
3 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	6334.2	±	3562.1	660%
3 nodes (Figure 11)	CUDA-SC	Service	ROS-SC	6755.5	±	4589.0	711%
3 nodes (Figure 12)	CUDA-SC	Topic	ROS-ZC	1087.6	±	39.4	31%
3 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	862.7	±	9.7	4%
3 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	9408.1	±	213.0	11%
5 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	12,234.2	±	3487.9	1369%
5 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	1737.1	±	9.1	109%
5 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	11,381.6	±	65.6	34%

In Table 2, the first two rows show how a *standard* implementation of the ROS protocol (i.e., ROS-SC) decreases the overall performance to unacceptable levels, for both services and topics ROS mechanisms. The ROS-ZC standard shows noticeable improvements compared to ROS-SC, by reducing the overhead by a factor of $\approx$100. The proposed ROS-SHM-ZC improves even further by reducing the overhead down to 2% and 11% when compared to CUDA-SC and CUDA-ZC, respectively.

Moving from two to three nodes, we found a performance improvement by combining CUDA-SC with ROS-SC in both topics and services. Although there is a slight overhead caused by the addition of the third node, these models lead to better overall performance due to the easier synchronization between nodes. The standard deviation shows high variance between results, suggesting a communication bottleneck that can be exacerbated by the system network conditions, outside of the programmer's control, even in *localhost*. This communication bottleneck is greatly reduced in the ROS-ZC and ROS-SHM-ZC configurations, thanks to the reduced size of the messages. The third node overhead still reduces the performance when compared to two nodes. Overall, the difference between two nodes and three nodes in the zero-copy configurations is negligible.

The same considerations hold for the five node architecture, which also proves to be very costly due to the additional nodes. Nonetheless, it is still better than the two nodes ROS-SC configuration, overhead-wise. In this instance, for the sake of clarity, the only reported solutions are the proposed ROS-SHM-ZC. The obtained overhead, while not negligible, is still limited, especially in the CUDA-ZC configuration.

Table 3. Results for the Concurrent Benchmark on the NVIDIA Jetson TX2. The reference execution time for CUDA-SC is 1053.0 ms and 1316.1 ms for CUDA-ZC.

Arch.	GPU Copy	ROS Type	ROS Copy	Time (ms)		SD (ms)	Overhead
2 nodes (Figure 6)	CUDA-SC	Topic	ROS-SC	4666.0	±	68.9	343%
2 nodes (Figure 7)	CUDA-SC	Service	ROS-SC	4930.0	±	73.3	368%
2 nodes (Figure 8)	CUDA-SC	Topic	ROS-ZC	1584.6	±	46.3	50%
2 nodes (Figure 8)	CUDA-ZC	Topic	ROS-ZC	1910.6	±	73.1	45%
2 nodes (Figure 9)	CUDA-SC	Topic	ROS-SHM-ZC	1046.1	±	82.3	−1%
2 nodes (Figure 10)	CUDA-ZC	Topic	ROS-SHM-ZC	1203.2	±	47.2	−9%
3 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	8244.6	±	2938.4	683%
3 nodes (Figure 11)	CUDA-SC	Service	ROS-SC	8505.7	±	767.6	708%
3 nodes (Figure 12)	CUDA-SC	Topic	ROS-ZC	2261.7	±	47.1	115%
3 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	997.9	±	75.6	−5%
3 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	1172.5	±	41.5	−11%
5 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	41,773.4	±	12,118.6	3867%
5 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	1923.	±	89.7	131%
5 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	1978.3	±	131.9	50%

Considering the concurrent benchmark on the Jetson TX2 (Table 3), we found performance results similar to the cache benchmark, but with lower overall overheads in the two nodes ROS-SC configurations. In this benchmark, the service mechanism of ROS is consistently slower than the topics mechanism. Worthy of interest are the speed-ups obtained with ROS-SHM-ZC. With CUDA-SC they are within a margin of error at −1% with two nodes and −5% with three nodes. With CUDA-ZC they are more significant at −9% and −11% with two and three nodes, respectively. This is caused by the caches not being disabled on the CPU node. As the hardware is not I/O coherent, we had to manually handle the consistency of the data, and because the cache utilization is low but still present, the CPU computations have better performance. This allows for an improvement when compared to the original CUDA-ZC solution of Table 1. Nonetheless, this also means that while there should be no copies in these two configurations, the combination of the ROS mechanisms with the CUDA communication model (CUDA-ZC) actually forces the creation of explicit data copies. One from CPU to iGPU and one in the opposite direction. These copies are responsible for the loss in performance when comparing ROS-SHM-ZC + CUDA-ZC to ROS-SHM-ZC + CUDA-SC.

Considering the Xavier device, (Tables 4 and 5), that ROS-SC model is much slower compared to the reference performance. Services are consistently slower than topics. ROS-ZC is faster than ROS-SC by a wide margin. It is also important to note that, for ROS-SHM-ZC, there are no negative overheads on the Xavier. This is due to the hardware I/O coherency, which already extrapolates the maximum performance for CUDA-ZC and the manual handling of the coherency does not lead to performance improvements. In both benchmarks, the three node configurations for ROS-SHM-ZC show higher overhead compared to the two node variants, highlighting the higher cost of the third node and, thus, leading to a significant performance loss when compared to the optimal performance of the native configuration.

Table 4. Results for the Cache Benchmark on NVIDIA Jetson Xavier. The reference execution time for CUDA-SC is 207.7 ms and 244.7 ms for CUDA-ZC.

Arch.	GPU Copy	ROS Type	ROS Copy	Time (ms)		SD (ms)	Overhead
2 nodes (Figure 6)	CUDA-SC	Topic	ROS-SC	8371.3	±	3210.8	3930%
2 nodes (Figure 7)	CUDA-SC	Service	ROS-SC	10,385.2	±	3423.4	4900%
2 nodes (Figure 8)	CUDA-SC	Topic	ROS-ZC	338.1	±	48.8	63%
2 nodes (Figure 8)	CUDA-ZC	Topic	ROS-ZC	653.6	±	35.4	167%
2 nodes (Figure 9)	CUDA-SC	Topic	ROS-SHM-ZC	230.0	±	15.5	11%
2 nodes (Figure 10)	CUDA-ZC	Topic	ROS-SHM-ZC	251.9	±	25.3	3%
3 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	5139.1	±	3484.6	2374%
3 nodes (Figure 11)	CUDA-SC	Service	ROS-SC	6744.4	±	4367.4	3147%
3 nodes (Figure 12)	CUDA-SC	Topic	ROS-ZC	333.9	±	30.5	61%
3 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	236.9	±	13.1	14%
3 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	400.9	±	20.8	64%
5 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	7466.4	±	5225.2	3495%
5 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	492.5	±	15.6	137%
5 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	847.6	±	36.4	246%

Table 5. Results for the concurrent benchmark on the NVIDIA Jetson Xavier. The reference execution time for CUDA-SC is 381.2 ms and 256.9 ms for CUDA-ZC.

Arch.	GPU Copy	ROS Type	ROS Copy	Time (ms)		SD (ms)	Overhead
2 nodes (Figure 6)	CUDA-SC	Topic	ROS-SC	3501.0	±	3052.9	818%
2 nodes (Figure 7)	CUDA-SC	Service	ROS-SC	5307.0	±	5841.7	1292%
2 nodes (Figure 8)	CUDA-SC	Topic	ROS-ZC	658.5	±	41.7	73%
2 nodes (Figure 8)	CUDA-ZC	Topic	ROS-ZC	672.4	±	34.9	162%
2 nodes (Figure 9)	CUDA-SC	Topic	ROS-SHM-ZC	403.3	±	43.0	6%
2 nodes (Figure 10)	CUDA-ZC	Topic	ROS-SHM-ZC	266.7	±	23.4	4%
3 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	17,054.5	±	5763.9	4374%
3 nodes (Figure 11)	CUDA-SC	Service	ROS-SC	25,033.2	±	14,269.0	6467%
3 nodes (Figure 12)	CUDA-SC	Topic	ROS-ZC	655.7	±	42.1	72%
3 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	408.0	±	56.9	7%
3 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	451.3	±	62.6	76%
5 nodes (Figure 11)	CUDA-SC	Topic	ROS-SC	43,808.4	±	8735.0	11,392%
5 nodes (Figure 13)	CUDA-SC	Topic	ROS-SHM-ZC	761.9	±	86.5	100%
5 nodes (Figure 13)	CUDA-ZC	Topic	ROS-SHM-ZC	859.0	±	66.5	234%

6. Conclusions

In this article we presented different techniques to efficiently implement CPU–iGPU communication that complies to the ROS standard. We showed that a direct porting of the most widespread communication protocols to the standard ROS can lead to up to 5000% overhead. We presented an analysis to show that each different technique provides extremely different performance depending on both the application and the hardware device characteristics. As a consequence, the analysis allows programmers to select the best technique to implement ROS-compliant CPU–iGPU communication protocols by fully taking advantage of the CPU–iGPU communication architecture provided by the embedded device, by applying different mechanisms included in the ROS standard, and by considering different communication scenario, from two to many ROS nodes.

Author Contributions: Conceptualization, N.B.; funding acquisition, N.B.; investigation, M.D.M. and S.A.; methodology, M.D.M., F.L. and S.A.; project administration, N.B.; software, M.D.M.; supervision, N.B.; validation, F.L.; writing—original draft, M.D.M., F.L., E.M., M.B. and S.A.; writing—review and editing, N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the project Dipartimenti di Eccellenza 2018–2022 funded by the Italian Ministry of Education, Univ. and Research (MIUR).

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mittal, S. A Survey of Techniques for Managing and Leveraging Caches in GPUs. *J. Circuits Syst. Comput.* **2014**, *23*, 1430002. [CrossRef]
2. Nvidia Inc. Nvidia Tootlkit Documentation, Unified Memory. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd (accessed on 27 March 2021).
3. Fickenscher, J.; Reinhart, S.; Hannig, F.; Teich, J.; Bouzouraa, M. Convoy tracking for ADAS on embedded GPUs. In Proceedings of the IEEE Intelligent Vehicles Symposium, Los Angeles, CA, USA, 11–14 June 2017; pp. 959–965.
4. Wang, X.; Huang, K.; Knoll, A. Performance Optimisation of Parallelized ADAS Applications in FPGA-GPU Heterogeneous Systems: A Case Study with Lane Detection. *IEEE Trans. Intell. Veh.* **2019**, *4*, 519–531. [CrossRef]
5. Otterness, N.; Yang, M.; Rust, S.; Park, E.; Anderson, J.; Smith, F.; Berg, A.; Wang, S. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, Pittsburgh, PA, USA, 18–21 April 2017; pp. 353–363.
6. Nvidia Inc. Jetson AGX Xavier and the New Era of Autonomous Machines. Available online: https://info.nvidia.com/emea-jetson-xavier-and-the-new-era-of-autonomous-machines-reg-page.html (accessed on 27 March 2021).
7. Lumpp, F.; Patel, H.; Bombieri, N. A Framework for OptimizingCPU-iGPU Communication on Embedded Platforms. In Proceedings of the ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 11–15 July 2021.
8. Lee, E.; Seshia, S. *Introduction to Embedded Systems—A Cyber-Physical Systems Approach*; MIT Press: Cambridge, MA, USA, 2015.
9. Adam, K.; Butting, A.; Heim, R.; Kautz, O.; Rumpe, B.; Wortmann, A. Model-driven separation of concerns for service robotics. In *DSM 2016, Proceedings of the International Workshop on Domain-Specific Modeling, Amsterdam, The Netherlands, 30 October 2016*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 22–27.
10. Hammoudeh Garcia, N.; Deval, L.; Lüdtke, M.; Santos, A.; Kahl, B.; Bordignon, M. Bootstrapping MDE Development from ROS Manual Code—Part 2: Model Generation. In Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019), Munich, Germany, 15–20 September 2019; pp. 95–105.
11. Bardaro, G.; Semprebon, A.; Matteucci, M. A use case in model-based robot development using AADL and ROS. In Proceedings of the 1st International Workshop on Robotics Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 9–16.
12. Wenger, M.; Eisenmenger, W.; Neugschwandtner, G.; Schneider, B.; Zoitl, A. A model based engineering tool for ROS component compositioning, configuration and generation of deployment information. In Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 6–9 September 2016.
13. Neis, P.; Wehrmeister, M.; Mendes, M. Model Driven Software Engineering of Power Systems Applications: Literature Review and Trends. *IEEE Access* **2019**, *7*, 177761–177773. [CrossRef]
14. Estévez, E.; Sánchez-García, A.; Gámez-García, J.; Gómez-Ortega, J.; Satorres-Martínez, S. A novel model-driven approach to support development cycle of robotic systems. *Int. J. Adv. Manuf. Technol.* **2016**, *82*, 737–751. [CrossRef]
15. Open Source Robotics Foundation. Robot Operating System. Available online: http://www.ros.org/ (accessed on 27 March 2021).
16. Yang, J.; Thomas, A.G.; Singh, S.; Baldi, S.; Wang, X. A Semi-Physical Platform for Guidance and Formations of Fixed-Wing Unmanned Aerial Vehicles. *Sensors* **2020**, *20*, 1136. [CrossRef] [PubMed]
17. Martin, J.; Casquero, O.; Fortes, B.; Marcos, M. A Generic Multi-Layer Architecture Based on ROS-JADE Integration for Autonomous Transport Vehicles. *Sensors* **2019**, *19*, 69. [CrossRef]
18. Kato, S.; Tokunaga, S.; Maruyama, Y.; Maeda, S.; Hirabayashi, M.; Kitsukawa, Y.; Monrroy, A.; Ando, T.; Fujii, Y.; Azumi, T. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In Proceedings of the 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), Porto, Portugal, 11–13 April 2018; pp. 287–296. [CrossRef]
19. Open Source Robotics Foundation. ROS 2 Documentation. Available online: https://docs.ros.org/en/dashing/index.html (accessed on 27 March 2021).
20. Open Source Robotics Foundation. ROS Wiki. Available online: http://wiki.ros.org/ (accessed on 27 March 2021).
21. Open Source Robotics Foundation. Efficient Intra-Process Communication. Available online: https://docs.ros.org/en/foxy/Tutorials/Intra-Process-Communication.html (accessed on 27 March 2021).