


RESEARCH

Open Access



Environment for integration of distributed heterogeneous computing systems

Thiago W. B. Silva², Daniel C. Morais¹, Halamo G. R. Andrade¹, Antonio M. N. Lima², Elmar U. K. Melcher² and Alisson V. Brito^{1*} 

Abstract

Connecting multiple and heterogeneous hardware devices to solve problems raises some challenges, especially in terms of interoperability and communications management. A distributed solution may offer many advantages, like easy use of dispersed resources in a network and potential increase in processing power and data transfer speed. However, integrating devices from different architectures might not be an easy task. This work deals with the synchronization of heterogeneous and distributed hardware devices. For this purpose, a loosely coupled computing platform named Virtual Bus is presented as main contribution of this work. In order to provide interoperability with legacy systems, the IEEE 1516 standard (denoted HLA - High Level Architecture) is used. As proof of concept, Virtual Bus was used to integrate three different computing architectures, a multi-core CPU, a GPU and a board with an Altera FPGA and an ARM processor, which execute a remote image processing application that requires a communication between the devices. All components are managed by Virtual Bus. This proposal simplifies the coding efforts to integrate heterogeneous distributed devices and results demonstrated the successful data exchanging and synchronization among all devices, proving its feasibility.

Keywords: Distributed computing, Heterogeneous computing, Middleware

1 Introduction

Integrating heterogeneous devices allows to raise the processing capacity without, necessarily, having a centralized control on a single device. To improve performance and increase cost-effectiveness, the processing tasks can be, normally, distributed. However, the integration of diverse devices demands a reliable communication, which is not an easy task, needing a mechanism that manages and synchronizes the members' messages. Building an environment to manage the exchange of data is even more difficult, because problems may arise from the integration of different devices.

The integration of computing systems (software and hardware) allows to create a System of Systems (SoS). Without a careful management there is high probability to instability and difficulties. According to [1], two systems can be considered stable when working individually, but nothing can be said about their operation at the time they are operating in an integrated manner. There are two

major problems: to split processing between all involved, dividing a task in subtasks to be processed by members; and assign a specific predefined task to each member in the system. A possible solution is to use a distributed communication architecture that allows the data exchange synchronously between the system's components.

A case study which puts the synchronization problem into evidence is presented in this paper. This problem appeared during the research when trying to verify some specific functionalities working with distributed systems in a functional verification setup. The main problems arise in the synchronization of the messages exchanged by the components. There was an inconsistency with regard to the way in which the components expected to exchange information. It often generated a certain rework and some communication problems during the previous experiments. Thus, this work aims at synchronizing communication of heterogeneous systems. Another key issue addressed in this work is the challenge of integrating legacy codes written in different languages for heterogeneous hardware architectures. The proposed solution provides a distributed computing platform with an API

*Correspondence: alisson@ci.ufpb.br

¹Universidade Federal da Paraíba (UFPB), Joao Pessoa, Brazil

Full list of author information is available at the end of the article

of high level functions for data exchanging and synchronization, independent of languages and architectures. To achieve this aim, our solution is based on the IEEE 1516 standard High Level Architecture (HLA) [2] as communication and synchronization platform.

HLA is a specification of a software architecture which has the purpose of facilitating intercommunication between distributed heterogeneous systems, mainly simulations, and allows the division of tasks among members [3]. This standard is a general-purpose architecture defined by the Defense Modeling and Simulation Office (DMSO) and designed to use a wide number of different types of simulators [2]. In this paper, HLA is used in an innovative way to provide interoperability of distributed heterogeneous hardware devices, instead of only simulations.

One of the possibilities proposed by the HLA specification is the use of diverse applications to compose a heterogeneous co-simulation. Therefore, it is feasible to build a computing platform based on the integration of heterogeneous devices and properly manage tasks to accelerate processing.

The purpose of this work is to create a platform that simplifies the intercommunication of distributed heterogeneous devices (composed of hardware and software). So, the main contribution of this work is the development of a platform to integrate heterogeneous computing devices (independent of architecture) in a loosely coupled way. As already being introduced, initially this work started from the idea of building a middleware to intermediate distributed devices to perform functional verification of components developed in the laboratory. Then, we decided to build a more general purpose software to abstract the underlying distributed architecture, instead of doing a specific solution. For this, the HLA standard for inter-operation among those systems was used, and a library was developed to unify the way of programming communication and synchronization. In previous work, HLA has been used to integrate circuit simulation tools for functional verification and power consumption estimation. In that case, different hardware architectures were simulated, but no physical devices were integrated [4, 5].

The HLA supports our implementation of a platform that emulates a bus, here named Virtual Bus. This paper also explores parallel computing in order to allow that multiples processing elements available in distributed devices can be used independent of their architectures. Virtual Bus is presented to programmers as an API with basic functions for reading and writing data to the bus, check available data and do synchronization.

As proof of concept, Virtual Bus was used to integrate three different computing architectures in a single platform: a multi-core CPU, a GPU and a System-on-Chip composed by a FPGA and an ARM processor. An example

was developed running a remote image processing application that requires communication between the devices. The usage of Virtual Bus permits to reduce the number of code lines necessary to integrate all components. Without Virtual Bus, it would be necessary to write at least about 1000 lines of code for each one, making it impractical as the number of components increases. With the Virtual Bus, it only takes a couple of lines to instantiate Virtual Bus and the Federates might be reused always when necessary. The platform may be extended to other architectures and to more devices in future works.

This paper is organized as follows. In the following section, related Works involving heterogeneous systems are presented. Section 3 gives a brief explanation of HLA and other background details. Then, in Section 4 the proposed platform that intercommunicates heterogeneous systems, the Virtual Bus itself, is addressed. Section 5 presents the methodology of the proposed experiments. The results of computational experiments are exposed in Section 6. Finally, a conclusion and perspectives are presented in Section 7.

2 Related work

This section presents some discussion about relevant aspects in related papers, i.e. works involving technologies with integration of distributed systems and heterogeneous hardware. Table 1 highlights the pros and cons of the related work compared to our approach.

In [6] a model that simulates a heterogeneous system controlled by Ptolemy II is presented. The major contribution of this work is the integration of different simulators with Ptolemy such as Simulink to model building automation systems. In our work, we do not use simulation to abstract heterogeneous hardware, but we use a distributed simulation platform, based on HLA and adapted to provide interoperability among heterogeneous hardware platforms.

In contrast with our approach, the work in [7] presents a programming model for modeling distributed systems, but it does not allow the execution of such systems in a distributed manner. This hinders the scalability of those systems. Other works use the concept of heterogeneous distributed systems to provide the connection of multiple systems. As the authors of [8], who propose a networked virtual platform to develop simulatable models of large-scale heterogeneous systems and support the programming of embedded applications. Different to our work, the contribution of that paper is the simulation of a system that includes processors and peripherals and uses binary translation to simulate the target binary code on top of a host instruction-set architecture. The integration of different hardware architectures in a distributed way is not considered.

Table 1 Comparison between different strategies

Papers	Comm. & sync.	Architectures	Application	Pros.	Cons.
Eidson et al. [7]	IEEE 1588 Network Time Protocol	Luminary Micro 8962	cyber-physical systems	A programming model for modeling distributed systems	Not allow the execution in distributed form of such systems
Jung et al. [8]	Developed by the authors	Any	Embedded devices communicating with cloud-computing server	Possibility to run real applications quickly without deploying a devices.	The model works only with virtual machines.
Vaubourg et al. [9]	Based on DEVS platform	Any	Simulation	The usage the TCP/IP network makes the possibility to work with a large scale of components.	Each component must be configured individually.
Van Tran et al. [10]	HLA	Any	Simulation	Enables to coordinate several parallel simulations as a distributed simulation system.	Focus on simulation, do not integrate heterogeneous architectures.
Gervais et al. [12]	HLA	Any	Simulation	This paper addresses the problem of achieving real-time performances with HLA.	It requires greater knowledge regarding the HLA.
Awais et al. [13]	FMI and HLA	Any	Systems embodying embedded software.	Enable a utilization of simulating multidisciplinary applications with several components of various types.	The solution is targeted only to embedded systems.
García-Valls et al. [15]	DDS	Raspberry Pi	Integration of DDS and Raspberry Pi	Analyzes the temporal behavior of the connection among embedded computers and servers in the context.	Focus on simulation, do not integrate heterogeneous architectures.
García-Valls and Calva-Urrego [16]	Developed by the authors	Multicore	Middleware developed	A middleware with high degree of integration with the hardware platform.	Focus on simulation, do not integrate heterogeneous architectures.
García-Valls et al. [18]	IRMOS and RT-Xen	Virtual machines	Simulation	Reconfigurable middleware for real-time distributed services-based systems.	The model works only with virtual machines.
Park e Min [19]	HLA and DDS	Any	Gateway/ Middleware services into HLA	Integrates HLA and DDS middleware and allows network control for large scale of distributed simulation systems	Not concern distributed RTI structure to give a solution for bottleneck problem of HLA.
Brito et al. [20]	HLA	Any	Simulation	Development and evaluation of a distributed simulation platform of heterogeneous simulators	Delay on synchronization
Our work	Virtual Bus over HLA/RTI	Any	Distributed computing	Facility to send and receive data without the need to understand the HLA protocol.	In some applications, a large number of Federates could possibly decrease performance.

A network solution is also suggested in [9], which proposes the integration of TCP/IP network simulators into a Discrete Event (DE) co-simulation platform. The paper proposes splitting network topologies into several models and defining input/output ports inside existing models. Our work delegates the network management to HLA, which defines all operations necessary for data exchange and synchronization.

In [10], a mixed simulation is introduced to coordinate several parallel simulations as a distributed simulation system. The parallel simulations are conducted according to HLA. The HLA has been used as co-simulation bridge. The work exposed by [11] uses HLA to run a fight simulation of aircraft attacking air defense units. In [12] HLA is applied for real time aircraft simulation, to validate real time behavior on target computing platform. None of these works deals with the problem of integrating heterogeneous architectures in a unique computing platform. Their focus is on simulation, while our work focuses on heterogeneous distributed computing.

The work [13] proposes to use HLA as a master for Functional Mockup Interface (FMI) compatible simulation components. The main objective is to provide a generic and standalone master for the FMI, making FMI-based simulation components usable as plug-and-play components, on a variety of distributed environments including grids and clouds. It is related to our work due to its goal to create a distributed computing platform, but its main objective is the simulation of FMI models, while we focus on heterogeneous distributed architectures.

In [14] the authors replace the transport layer of HLA-based system by Data Distribution Service (DDS) communication. They present a combination of distributed HLA-based simulation with network control using DDS. The HLA and DDS are combined to form a unique middleware. It consists of service and network configuration and an API for interconnecting the data object between HLA and DDS. HLA-DDS does not only allow network controllable distributed simulation but also preserves existing HLA-based distributed simulation systems. The goal is to implement a bridge between HLA and DDS, while our work focuses on lower level integration, when different hardware architecture can be integrated.

DDS has also been used in [15] to manage the interaction between high computation power nodes and ARM-based embedded computers. In that work, a flexible library to create the communication using different underlying communication software is presented. The target system integrates heterogeneous nodes and base servers. Although our solution is built on HLA, another version could also be implemented over DDS. In future works, our solution may also use DDS in replacement of HLA for distributed and heterogeneous applications.

Our work brings a contribution regarding abstraction and intercommunication, but in the case of time-sensitive applications, the work of [16] is more specific. The authors propose a middleware with high degree of integration with the hardware platform, through the use of operating system calls to control the computing cores. However, our work proposes a more generic solution, independent of hardware architecture or operating system.

There are other works that investigate middlewares that supports composition of components, services and modules, with support to dynamic changes in real time [17, 18]. The authors propose reconfigurable middleware for real-time distributed services-based systems. However, our solution focuses on the integration of different hardware platforms in a unique environment in a loose-coupled way, not necessarily based on services nor components.

The authors in [19] propose a Gateway/Middleware High Level Architecture (HLA) implementation and the extra services that this implementation provides to a simulation. That paper contributes to incorporate Gateway/Middleware Services into HLA interface that is denoted, a Simulation Object Middleware Classes (SMOC) Gateway.

In our previous work, HLA is used to integrate five different simulations tools: Ptolemy II, SystemC, Omnet++, Veins, Stage and physical robots [20]. The idea is the development and evaluation of a distributed simulation platform of heterogeneous simulators. That work inspired the present work with the idea to extend HLA for not only simulations, but to general computing applications running in heterogeneous hardware architectures.

In our solution, OpenCL is used to explore parallel computing in multi-core CPU and in GPU, due to its versatility. Other works have also used OpenCL, to explore high-performance computing [21, 22], though it presents lower performance than CUDA solutions [22]. Therefore, the most advantage in using OpenCL in our context is the vast compatibility with heterogeneous hardware platforms.

In Table 1 the related works are compared with our work focusing on main contributions of this paper.

3 The high level architecture (HLA)

The High Level Architecture (HLA) is a standard of the Institute of Electrical and Electronic Engineers (IEEE), developed by Simulation Interoperability Standards Organization (SISO). Initially it was not an open standard, but it was later recognized and adopted by the Object Management Group (OMG) and IEEE [2].

There are several standards based on distributed computing, such as SIMNET, Distributed Interactive Simulation (DIS), Service Oriented Architecture (SOA), Data Distribution Service (DDS), HLA, among others. HLA

was chosen as standard to integrate distributed heterogeneous devices because it manages both, data and synchronization, and allows the interoperability and composition of the widest possible range of platforms. One of the most notable advantages of using HLA for this purpose is that it already has a trustworthy and widely used solution for time synchronization. There are also a large quantity of simulations and tools compatible with it (e.g. Matlab, Simulink, OMNet++, Ptolemy) which turns easier further applications with different tools.

HLA is not a software implementation, but a standard with diverse independent implementations, including some open-source, like CERTI [23] and Portico [24]. HLA is specified in three documents: the first deals with the general framework and main rules [2], the second deals with the specification of the interface between the simulator and the HLA [25] and the third is the model for data specification (OMT) transferred between the simulators [26].

The main HLA characteristics are defined under the leadership of the Defence Modelling and Simulation Office (DMSO) to support reuse and interoperability. Interoperability is a term that covers more than just send and receive data, it also allows multiple systems to work together. However, the systems must operate in such a way that they can achieve a goal together through collaboration.

3.1 Architecture

The main idea of the HLA is to provide a general purpose platform where the functionality of a system can be separated in distributed machines without loss of consistency. For this, it uses the Runtime Infrastructure (RTI), which manages data exchanging and centralize the control of a global time for synchronization among the Federates (see Fig. 1). This union of Federates through RTI is called Federation. Here we use the term HLA Time to distinguish it from local time of each Federate and it refers to a logical time and not a clock time.

To connect various Federates with RTI, two components must exist: one local RTI Ambassador (RTIA) and a global RTI Gateway (RTIG). RTIA defines the interface of the Federate with RTI, calling functions of RTIG for updating, reflection, transmitting and receiving data. RTIG is responsible for synchronization and data consistency. Messages among RTIG and RTIA are exchanged through a TCP/IP network in order to perform the services in a distributed manner. In this work, the HLA implementation CERTI [23] was used. CERTI is an open source implementation of HLA specification and is developed by its open source community and maintained by ONERA. This implementation supports HLA 1.3 specification and it is already used in robust co-simulations [4, 20, 27–29].

3.2 Interfaces

Federates do not communicate directly with other Federates. Each one is connected to the RTI, then they communicate with each other using only the services that the RTI provides. There is always an interface between the RTI and the Federates, and each member has a unique connection with the RTI.

The RTI provides an interface called RTI Ambassador and for each Federate an interface called Federate Ambassador must be implemented for communication with RTI, as presents in Fig. 1. Typically, RTI Gateway (RTIG) is provided by HLA implementations and developers must implement (or reuse) a Federate Ambassador for each system or device that will be part of the Federation. In this work, three Federate Ambassadors were developed, for GPU, SoC and Multi-core CPU. The Federate Ambassador has two main objectives: to exchange data through RTI, and to manage the synchronization with the RTI.

In our implementation, we use the available “publish and subscribe” communication mechanism provided by RTI. Messages are used to update values by calling the function *updateAttribute* of the Federate Ambassador. All updating is requested by a Federate to RTI, which

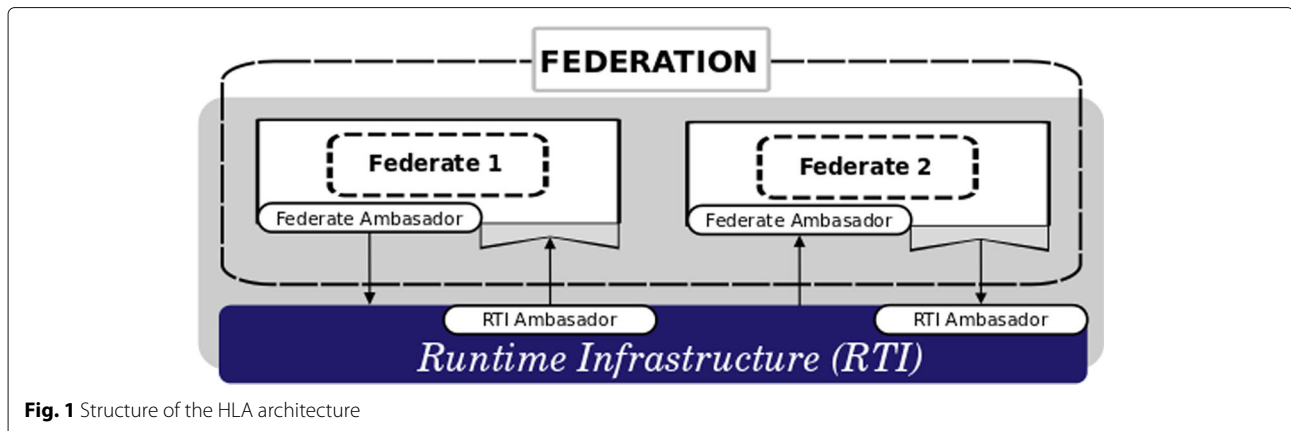


Fig. 1 Structure of the HLA architecture

propagates it, calling the function *reflectAttribute* of all Federate Ambassadors. Once it happens, our implementation of this function save the values into internal variables, and signalize that a new data have been received. This flag will remain on, until the *getReceivedData* function is called for reading.

3.3 Time management

To deliver the messages in a consistent order, the HLA has specific mechanisms of time management. They are associated with the idea of advancing time step, which is an abstraction of a global time to all Federation, which we call HLA time. RTI manages the advancing of HLA time to guarantee that each Federate will advance to next step only when all the others reach the same HLA time.

For this, the Federate Ambassador defines the *federateTime* and *advanceTime* functions. The first one is used to read to current global time (or HLA Time), and the second is to send an advancing time request to RTI. The Federate is blocked until RTI grants the time advancing. The grant will occur only when all registered Federates request the time advancing to the same point.

As a Federate communicates with each other through RTI, the data exchanging is performed in terms of interactions and objects. An interaction is the operation of sending data in one time-step, and objects are the data packets sent during an interaction. To initiate a Federation, it is necessary to start the RTI Gateway (RTIG) to allow all Federates to join the Federation. Updates of new messages are received when a Federate applies for an object. Therefore, all updates in those objects are reflected to those interested Federates. The Virtual Bus encapsulates both the request and the reflection of objects.

Each Federate knows its own internal logical time and can advance it following some policies. A Federate can be time-constrained, when the advance of local time is restricted by other Federates; can be time-regulating, in which the advance of its local time regulates other Federate; both or none. In this project the Virtual Bus configures the time management of all Federates to both, time-constrained and time-regulating.

4 The platform for distributed heterogeneous computing (Virtual Bus)

In this section is presented the proposed platform for distributed heterogeneous computing called Virtual Bus, which is responsible for sending and receiving data on the network and for allowing interoperability between multiple heterogeneous hardware platforms.

This work used the intercommunication standard HLA (see Section 3) as a middleware for communication between these platforms. Virtual Bus has the role of letting data exchange operations transparent to the user, providing an API over the HLA, without the user having to

perform its configuration explicitly. So, each device is a Federate that will communicate through the Virtual Bus.

Figure 2 presents the extensions of HLA proposed here (called Virtual Bus) implemented to turn the distributed computing more transparent. Different architectures as CPU, GPU, ARM and FPGA might be connected using the Virtual Bus, which is built on top of the CERTI/HLA environment.

To join a Federation, a Federate must call the *runFederate* function from Virtual Bus API, described in Code 1. This function creates an instance of RTI Ambassador, requesting the RTI to create the Federation if it does not exist, and create the Federate Ambassador. The actual Federate joins the Federation and signals the RTI that it is ready to run as soon as all other Federates reach the synchronization point called *READY_TO_RUN*. Finally, when a Federate calls the method *publishAndSubscribe()* the time policy is set and all interesting objects to receive and send updates are registered.

Code 1 Pseudocode of *runFederate* function

```

1  runFederate( char* federateName ) {
2  //Create Federation if doesn't exist
3  rtiamb = new RTI::RTIambassador();
4  rtiamb->createFederationExecution();
5
6  //Create the FederateAmbassador
7  fedamb = new FederateAmbassador(federateName)
8
9  rtiamb->joinFederationExecution( federateName
10 , fedamb );
11 rtiamb->synchronizationPoint( READY_TO_RUN );
12 publishAndSubscribe();
13 oHandle = registerObject();
14 }
```

Virtual Bus also offers in its API two functions for writing and reading data to facilitate communication between the Federates. The *writeData* function is responsible for sending data through Virtual Bus. The main logics of these functions are presented in Code 2. It creates an object to manipulate its attributes. All new values are set to these attributes, which are sent to the RTI together with the HLA local time of that Federate. As previously presented in section 3, Virtual Bus configures all Federates to Time Constrained and Time Regulating policies. This

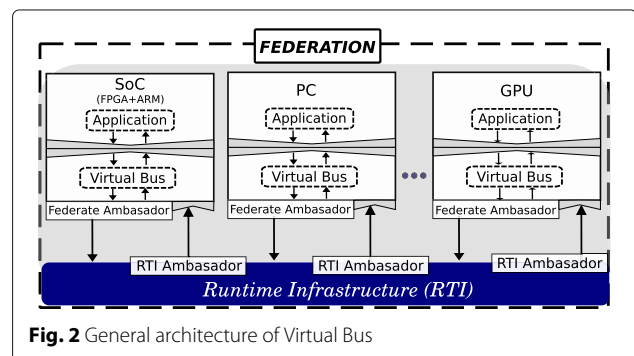


Fig. 2 General architecture of Virtual Bus

guarantees that a Federate will advance its local time to a specified global time (HLA time) only when all other Federates also reach a time equals or greater than that.

Code 2 Pseudocode of writeData function.

```

1  writeData(id, data){
2  attributes = new RTI::Attribute();
3  attributes ->add(id);
4  attributes ->add(data[0]);
5  attributes ->add(data[1]);
6  attributes ->add(data[2]);
7  ...
8  attributes ->add(data[N]);
9
10 //Get HLA time from Federate
11 time = fedamb->federateTime();
12
13 //Update value in RTI
14 rtiamb->updateValues(oHandle, *attributes,
15     time);
16
17 //Advance time
18 fedamb->advanceTime();
19 }
```

To explicitly request time advancing, Federates must call the *advanceTime* function from Virtual Bus API and wait for a granting message from RTI. Only when all Federates are granted, the global time of the Federation is advanced. Meanwhile, the Federate is blocked waiting for this granting. In Virtual Bus, the *advanceTime* is called always when the function *writeData* of Virtual Bus is called. It means each Federate advances its local time after each updating the values of the attributes registered to it. When every Federate advances its local time, RTI advances the global time and one cycle is completed.

To receive data, the Federate must use the *readData* function that returns the last received data from RTI. The function works as follows: if any data has been received, the Federate updates a flag to true. This flag can be checked by *hasReceivedData* function. If there is available data, it is returned, otherwise a null value is returned. The pseudo code of *readData* function is shown in Code 3.

Code 3 Pseudocode of readData.

```

1  Object readData(id, data){
2  if (fedamb->hasReceivedData(id)){
3  data = fedamb->getReceivedData(id, data);
4  return data;
5  }
6  else
7  return null;
8  }
```

The Virtual Bus works as illustrated in the Fig. 3. On the sender side, the *writeData* function is used to send data through RTI. Once this function has been called, the values are updated calling the *updateValues* function, which calls *updateAttributes*. In this step, each component of an array item is rearranged and passed to RTI Ambassador. The control of time and distribution of data is carried out by the RTI, calling the functions to synchronize all Federates (*waitSync*) and to distribute data among all registered Federates (*distributeData*).

On the receiver side, data is reflected into internal variables by the *reflectAttribute* function, called by the Federate Ambassador. This method calls *receivedData* to store data internally and set the flag *hasReceivedData* to true. A Federate in Virtual Bus is configured to check at each internal cycle if some new data was received by calling *readData* method. This method checks the flag and, if it is true, the data is returned to application.

Code 4 Data Object Model for Virtual Bus.

```

1  (FED
2  (Federation Test) (class VirtualBus
3  (attribute privilege)
4  (class RTIprivate)
5  (class port
6  (attribute id)
7  (attribute data0)
8  (attribute data1)
9  (attribute data2)
10 ...
11 (attribute dataN)) ) )
```

The format of the data exchanged by the Ambassadors is defined following the Object Model Template (OMT) of the HLA [26], which is specified in a file common to all Federates. In Virtual Bus, each object has attributes to identify the destination and origin of message, besides attributes of N data values. The size of N is set in advance depending on each configuration scenario. The description file for the Data Object Model for Virtual Bus is presented in Code 4.

The Virtual Bus offers a general propose API for distributed systems, and its use must be easy and simple. So, the only changes that are needed to integrate new projects, or even legacy codes, are to add some libraries from the CERTI HLA and to include the Virtual Bus package. As shown in the Fig. 4, the package contains basically the Virtual Bus Federate and Federate Ambassador (classes and interfaces). From this point, to use the Virtual Bus the only necessary functions to be called are: *runFederate*, *writeData* and *readData*. The first one is to initialize the Federate, the second one to send and the last one to receive data. The Federate Ambassador is referenced as a black box code, used by the Virtual Bus Federate and does not have to be called directly.

5 Experiment

The main idea of these experiments is to run some Federates that exchange data of different types and sizes. Therefore, the experiment was assembled with four Federates: the Sender Federate (running in a PC), the SoC Federate (ARM+FPGA), the Multi-core Federate and the GPU Federate. The Sender Federate sends images to the other Federates, that will process some operation on the images and return the result back to the Sender Federate. All Federates use the same implementation of Virtual Bus developed in C++.

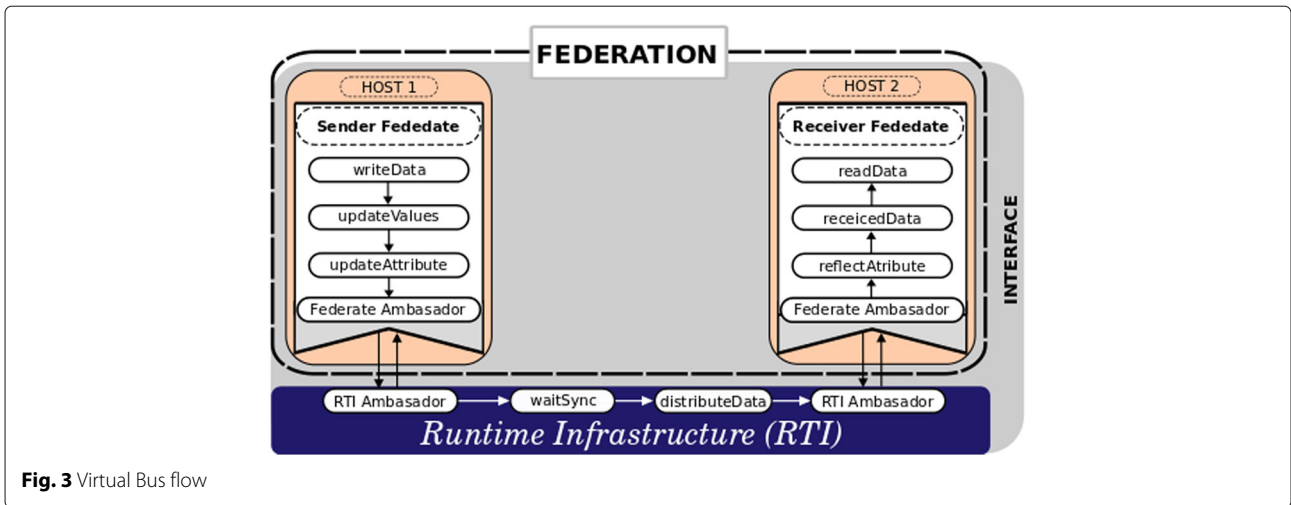


Fig. 3 Virtual Bus flow

To ease the manipulation of the images, the OpenCV framework was used in Sender and Multi-core Federates. OpenCV is a library used to manipulate and process images, originally developed by Intel (<http://opencv.org>). It was used in this work only for basic handling of pixels through its functions and data structures.

In this section is presented the configuration of the experiment and some lessons learned during the work. In Subsection 5.1 the list of equipment specifications can be found and how they are connected. The Subsection 5.2 shows the scenarios that were configured. Many data formats were tested and they are presented in Subsection 5.3. Then, it is given a more detailed description of how each Federate works in Subsections 5.4 and 5.5.

5.1 Equipments

In general, a Federation was configured composed by four computing machines, corresponding to the following Federates: the Sender Federate, the Multi-core Federate, the GPU Federate and the SoC Federate (ARM+FPGA). The basic configuration of each one is describe in Table 2. The Sender Federate is a desktop computer running Ubuntu 14.04 LTS. The SoC (ARM+FPGA) has an Altera Cyclone V SE SoC, which has a Cyclone V FPGA integrated with a

dual-core ARM Cortex A9 processor on a single chip, running Ubuntu 12.04 LTS. The GPU Federate uses a GeForce GT from NVidia, and was running Ubuntu 16.04 LTS. The multi-core Federate runs OpenSuse 13.2 Harlequin.

5.2 Scenarios

The experiment was divided in five scenarios as listed in Table 3. In the first scenario, the Sender Federate communicates only with the SoC. Following, it communicates separately with Multi-core and then with GPU Federate in scenarios 2 and 3, respectively. In scenario 4 the communication is done between the Sender, SoC and Multi-core Federates. Then, in the last scenario, the Multi-core Federate is replaced by the GPU Federate.

The idea in these scenarios is to test separately each Federate with the Sender Federate in scenarios 1 to 3, and later to integrate two Federates per experiment in scenarios 4 and 5. With this, it is possible to analyze the behavior of the Virtual Bus in separated cases.

Figure 5 gives an overview of how the devices are connected. The Sender Federate is in the left side of the figure. the Sender Federate. It is responsible to generate data to all other Federates and collect the results from them. In the right side of the same figure are the other Federates: SoC, where the ARM bridges the Virtual Bus with the FPGA, and the Multi-core and GPU Federates, which use OpenCL to interface the Virtual Bus with the parallel architecture.

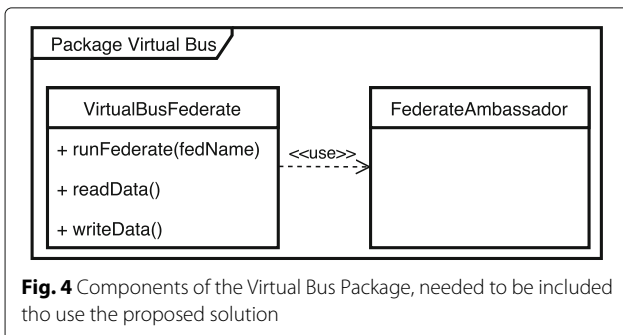


Fig. 4 Components of the Virtual Bus Package, needed to be included to use the proposed solution

Table 2 Equipment specification

Device	Configuration
Sender	Intel Celeron 430 with 2 GB SDRAM
SoC (ARM+FPGA)	DE1-SoC Board from Terasic
Multicore	Intel i3-4005U with 4 GB DDR3
GPU	GeForce GT 740M

Table 3 Scenarios used in the experiments

Scenario	Sender	SoC (ARM+FPGA)	Multi-core	GPU
1	X	X	-	-
2	X	-	X	-
3	X	-	-	X
4	X	X	X	-
5	X	X	-	X

5.3 Data configuration and exchanging

One of the contributions of this work is to improve the data transfer to an acceptable rate. In this subsection, we present some results in the development process with some details regarding the implementation of data exchanging in Virtual Bus. This discussion is more relevant in the cases where a considerable amount of data must be transferred, like an image, for example. In this experiment the following data exchanging strategies were used:

- one-by-one: pixels are sent one by one in each HLA message;
- multi-pixel: a group of N pixels are sent in N attributes, one attribute for each pixel;
- multi-pixel in one attribute: a group of N pixels is sent in one array attribute of N size.

Some formats for the messages were defined to improve the data exchanging in Virtual Bus. The overall format is presented in Fig. 6.

The field called *data* contains part of the image that follows in each message. During the experiments, a variation of sizes of the data in messages were experimented, which resulted into different data transfer methodologies. They essentially differ in the number of pixels per message and the way the pixel information is organized in attributes.

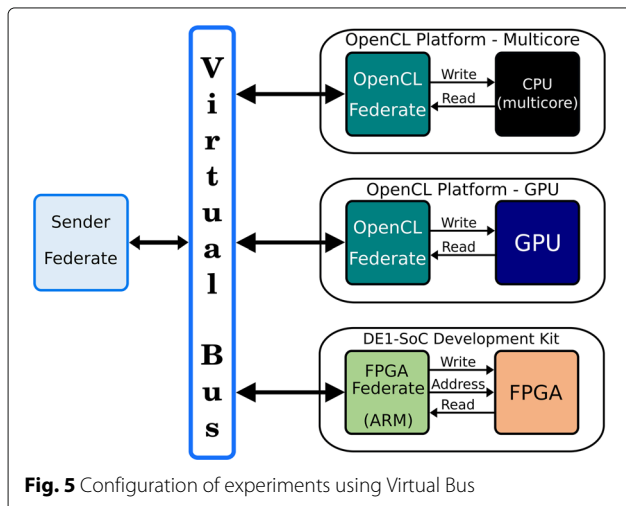


Fig. 5 Configuration of experiments using Virtual Bus



Fig. 6 General structure of messages

The first methodology, hereinafter referred as one-by-one, has been implemented to send one pixel per message. That is, to send an image, each message have the information of the source, plus the position of pixel (x and y) and the corresponding pixel data. Thus, the amount of messages is equal to the number of pixels in the image.

These messages were structured like presented in Fig. 7. For example, the source field is the ID of the Sender Federate, the address is the ID of the target Federate that must receive the message, the position x and y are the pixel coordinates being sent and the pixel_data field is the content of pixel itself.

In the second methodology of the experiment, called multi-pixels, it has been adopted the strategy of sending image information only in the first message, such as resolution and number of channels. And then the next messages carry only the pixels (multiple ones by messages). It also means a variation in the number of fields per message.

Remembering that the number of elements is equal to the number of pixels multiplied by the number of channels. For example, five pixels in a image of three channels (RGB) means fifteen data fields per message. The structure of the messages is presented in Fig. 8.

Based on the first message sent with the resolution information and number of channels, it is possible to manage the receipt of pixels. Hence, to send a complete image, this strategy produces the following number of messages: the number of pixels, times the number of channels, divided by the number of elements sent per message, adding yet the first message.

The last methodology to transfer the images, called multi-pixel in one attribute, is a variation of the second implementation. The structure of the message is the same as presented in Fig. 8, but here the HLA is used in a different way. Now, it adds multiple pixel content in only one field of HLA message attribute. Here the Object Model used in Virtual Bus (as shown in Code 4) is changed to use one unique data field of multiple elements. This field is managed by HLA as an array, thus the data of all pixels is encapsulated in a unique array type (also called data). In HLA, it means the RTI will try to send as maximum as data per TCP packet, instead of been limited by a fixed number of data fields.

As it is presented in next section, this methodology called **multi-pixel in one attribute** achieved the best performance. Therefore, this was the chosen approach to be used in the following experiments presented here.

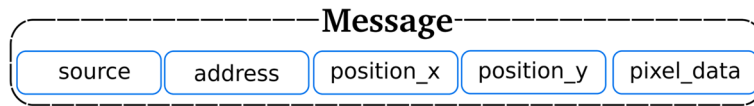


Fig. 7 Structure for one-by-one messages

As presented in next section, the aforementioned methodology called thereafter multi-pixel in one attribute provides the best performance. Consequently, this strategy has been used in the following experiments.

5.4 Sender federate

The Sender Federate is responsible for sending data to be processed by the other Federates and for receiving the results back. The Sender Federate opens an image file with resolution of 512 × 512 pixels and sends all the pixels. In practice, this image is a matrix of unsigned char elements. Because the image is colored, each pixel is represented in three channels, this is due to the RGB representation.

The data transmission using Virtual Bus turns the interactions between those Federates involved with the execution transparent. After the image is loaded into memory, each pixel is fragmented in a scalar format to be sent in bursts, repeated in a main loop. In the Code 5 the main logic is presented. Notice that the first action is to start up the RTI (lines 2-4). Following, the value of each channel in a pixel is stored in an array (lines 7-10). The next lines (14-21) check if all elements of image were already sent or if it must send the next line of the image matrix. Finally, the data array is sent to RTI (line 26). In the last lines (28-35), it receives the processed data from RTI and handle it according to which is the destination Federate.

This loop code is for sending data up to the value of NUMBER_OF_ELEMENTS_BURST variable is reached. For example, if 900 elements burst size is chosen, that means 300 pixels per message will be sent. So, the for structure has the stop condition because it buffers the 300 pixels in the variable called data, to send it subsequently. The x and y variables represent the coordinates of the pixel which is being accessed.

5.5 SoC integration with Virtual Bus

As proof of concept, the MD5 algorithm was implemented and executed in FPGA. MD5 is a hash function vastly used as a checksum to verify data integrity [29], which takes as input a message of arbitrary length with maximum of 512 bits and produces as output a fingerprint of 128 bits.

Code 5 Main loop in Sender Federate to send images

```

1  ...
2  VirtualBusFederate *federate;
3  federate = new VirtualBusFederate();
4  federate->runFederate( federateName );
5  ...
6  for(i=0; i<NUMBER_OF_ELEMENTS_BURST; i++) {
7    Vec3b s = image.at<Vec3b>(Point(x, y));
8    data[ position++ ] = s.val[0];
9    data[ position++ ] = s.val[1];
10   data[ position++ ] = s.val[2];
11
12   numberElemSent += 3;
13
14   if(++x == resolutionX) {
15     if(numberElemSent == totalElements) {
16       sentAllElements = true;
17       break;
18     } else {
19       //jump to next line
20       x = 0;
21       y++;
22     }
23   }
24 }
25 federate->writeData(Sender_ID, data);
26
27 if(federate->readData(src, data)) {
28   switch(src) {
29     case FPGA_ID :
30       //handles data from FPGA
31       ...
32     case PROCESSOR_ID :
33       //handles data from Multicore/GPU
34       ...
35   }

```

The input message should be an arbitrary and not negative integer.

The Code 6 shows how the Federate is implemented in the ARM processor. As an initial solution, a loop is proposed instead of using processor interrupts to check if any data is received. Once received, the function to calculate MD5 by the FPGA is called (line 9). The result is only sent to the Virtual Bus when the calculation is completed. This is controlled by a flag called received (line 11). When the result of the calculation is sent by the FPGA, then the four words is sent to Virtual Bus (line 12) and received by Sender Federate on the other side.

The communication between the ARM and FPGA in the software layer is made by the calculate_md5 function



Fig. 8 Example of multi-pixel message, transporting five pixels of three channels

Code 6 Logic of the Federate running on ARM

```

1  ...
2  // create and run the federate
3  VirtualBusFederate *federate;
4  federate = new VirtualBusFederate ();
5  federate->runFederate( federateName );
6
7  while(1){
8      if( federate->readData( source , data)){
9          calculate_md5( data , a , b , c , d);
10     }
11     if( received ) {
12         federate->writeData( federateName , a , b , c , d);
13         receive = false;
14     }
15 }
16 federate->finalize ();
17 ...

```

in line 8 at Code 6. In ARM, this communication is done through written records. This is configured with the Qsys framework, which maps the FPGA as a peripheral device of the ARM processor. The MD5 was implemented as a FSM which receives a sequence of 512 bits, separated in 16 blocks of 4 bytes each.

The Cyclone V SE SoC has a physical limitation that does not allow the transmission of 512 bits in one clock cycle. So, we have created a wrapper in Verilog to connect the MD5 code (in FPGA) with the ARM. This logic splits the transfer between FPGA and ARM in transfers of 32 bits, until the 512 bits are transferred (see Code 7). Thus, the input signals *inwdata* and *inaddr* and output signal *outrdata* are mapped in the ARM registers and can be easily accessed from the software layer.

Code 7 Verilog MD5 block: a wrapper to connect MD5 into ARM

```

1  \begin{lstlisting}[]
2  module md5_wr ( clk ,
3                reset ,
4                in_wdata ,
5                in_addr ,
6                outrdata );
7
8  //Define Input/Output
9  input wire    clk;
10 input wire    reset;
11 input wire [31:0] in_wdata;
12 input wire [63:0] in_addr;
13 output reg [31:0] outrdata;
14 ...

```

The wrapper receives data from ARM via *inwdata*, and store it in a register bank at address provided by *inaddr*. To read the result from MD5, it is necessary to wait 63 positive clock edges, then set *inaddr* to the address that holds the results and read *outrdata*. For reading and writing 32-bit words are used, while for addressing 64 bits are used.

The registers from 0 to 15 (4 bytes each) are used for transmitting parts of the message. After sending the message completely, the least significant bit from register 16 is set in order to turn the MD5 block available. So, after 64 clock cycles the result is stored in registers 32 to 35.

Finally, *inaddr* is set to indicate the address of registers and their values are returned via *outrdata*.

5.6 Federates based on OpenCL

In our implementation, two Federates are based on OpenCL, the Multi-core Federate and the GPU Federate. Both of them work in similar ways. They receive all data of image from Sender Federate in the same way the other Federates, then they build the image matrix in the device memory and an OpenCL kernel is initialized. So, they execute a mask operation in the image. It consists on recalculating all the image pixels by applying the Eq. 1.

$$I(i,j) = 5 * I(i,j) - [I(i-1,j) + I(i+1,j) + I(i,j-1) + I(i,j+1)] \Leftrightarrow I(i,j) * M, \quad (1)$$

$$I(i,j) * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (2)$$

The Eq. 1 was obtained by multiplying each image element by the mask matrix, as shown in Eq. 2. This calculation adjusts each pixel value based on how much influence the current and the neighbor pixels have.

To reach a satisfactory portability of the kernel between the diverse hardwares, some calculations were done to adjust the runtime environment of the OpenCL. When using OpenCL is important to calculate properly the number of work-groups [30, 31]. For this implementation, it was taking into account the number of elements to be calculated based on the resolution of the image, the number of cores of the current architecture and the number of compute units of the processor. This last two parameters are based on the values returned by some OpenCL functions appropriate for querying hardware attributes.

Given the low degree of complexity of the kernel in this experiment, only this information is necessary to calculate the required number of threads. Dividing them into work-groups with appropriate amounts according to the number of compute units.

In the experiment involving Multi-core CPU and GPU Federates, the following steps were executed:

1. Sender Federate reads an image and shows it on the screen;
2. Sender Federate sends the image to the Multi-core/GPU Federate;
3. Multi-core/GPU Federate receives the image and displays it on the screen for a subjective integrity check;
4. Multi-core/GPU performs the processing of OpenCL kernel;
5. Multi-core/GPU shows the resulting processed image;

6. Multi-core/GPU sends a response to the Sender Federate;
7. Sender Federate receives the response and displays it on the screen.

6 Results

The results presented in this section refer to an analysis of the data transfer approaches, presented in Subsection 5.3, and then some results from the experiments discussed in Section 5, and specifically presented in Subsection 5.2. In Subsection 6.1 are presented the data exchanging results, and in the next subsections are presented the overall results for the different experiment scenarios.

6.1 Data exchanging analysis

Table 4 presents the time and throughput to transmit an image from the Sender Federate to other Federates via an Ethernet LAN network. The Lena image used in the experiments is presented in Fig. 9, with the resolution of 512×512 in RGB (a matrix of 786,432 unsigned char elements, or 768 KB).

In the first experiment, pixels are sent one by one in each HLA message. A multi-pixel approach is used in experiment 2, where 15 pixels are sent in 15 attributes (HLA), one attributes for each pixel. In experiment 3 the same approach is executed, but now with 100 pixels. In experiments 4 and 5, a group of 100 and 300 pixels, respectively, are sent in arrays of the same size. In these two last experiments the time is much lesser because HLA tries always to send to complete array in a unique message. This experiment was important to evaluate the impact of different approaches in HLA to organize data in messages.

It is important to note that this throughput average is based only on the image data sent and received (payload), not including the traffic of control messages sent by the CERTI RTI implementation. This gives an idea of the necessary time to transfer data via Virtual Bus. Thus, it turns more evident the Virtual Bus capacity of sending the image from one Federate to another.

The first line of the table contains the values from the one-by-one experiment, line 2 and line 3 refers to results in the multi-pixel experiments, where there is one



Fig. 9 The Lena image used in the experiments

attribute for each element to be sent. Finally, line 4 and 5 are the results for multi-pixel in one attribute, that sends multiple pixels in a single HLA attribute. The last column named “speedup” presents an overall speedup of each throughput result in comparison with one-by-one approach.

Comparing the line 3 and 4, the same number of pixels per message was sent, but with different transmission approaches. In this case occurred a time reduction and an increase in throughput when more pixels were transmitted by message.

The increase of the number of attributes from one to 15 pixels per message, respectively (experiments 1 and 2 in Table 4), brought a speedup of 9.5 times. When transferring 100 pixels per message (experiments 3), the speedup was 16.5 times, demonstrating a smooth increasing. And the highest speedups were achieved when the multiple data was encapsulated in a unique array attribute, reaching speedups of 20 and 317.7 times (experiments 4 and 5, respectively), demonstrating a exponential increasing.

Table 4 Transfer times of Lena image with 786,432 elements

#	Experiment	Time	Throughput average	Speedup
1	One-by-one	72 s	87 Kbps	–
2	15-pixels	7.8 s	800 Kbps	9.5 X
3	100-pixels	4.4 s	1.4 Mbps	16.5 X
4	100-px/one attrib.	1.7 s	3.7 Mbps	20.0 X
5	300-px/one attrib.	234 ms	27 Mbps	317.7 X

Table 5 Processing time and total time of Lena image with 786,432 elements by GPU Federate

#	Experiment	Processing time	Total time
1	One-by-one	281 ms	72 s
2	15-pixels	281 ms	7.8 s
3	100-pixels	281 ms	4.4 s
4	100-px/one attrib.	281 ms	1.7 s
5	300-px/one attrib.	281 ms	515 ms

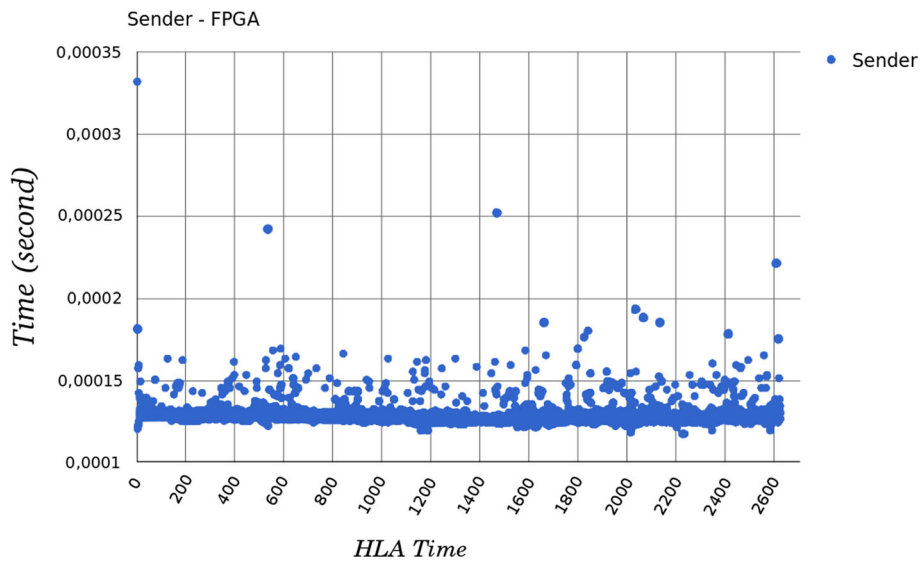


Fig. 10 Sender Federate activity during the transition to SoC Federate

This result demonstrates the improvement obtained from different methodologies addressed in the proposed environment. The speedup is presented comparing the different configurations with the most simple one, where only one pixel is transmitted per simulation cycle. This provides data for comparisons in future work to assist the choice of which HLA configuration is more appropriate when using Virtual Bus. In this experiments, we demonstrated that for applications where large amounts of data must be transferred, the most appropriate approach

is to transfer multiple data in a unique HLA array type, like in experiments 4 and 5.

With OpenCL it was possible to implement a component that allows the use of heterogeneous hardware platforms integrated to Virtual Bus. This enables the use of both multi-core CPU, and GPU. It was also possible to adaptively manage the number of work groups. This number is calculated dynamically according to the image resolution and to the number of cores, among other device-specific features. This calculation made pos-

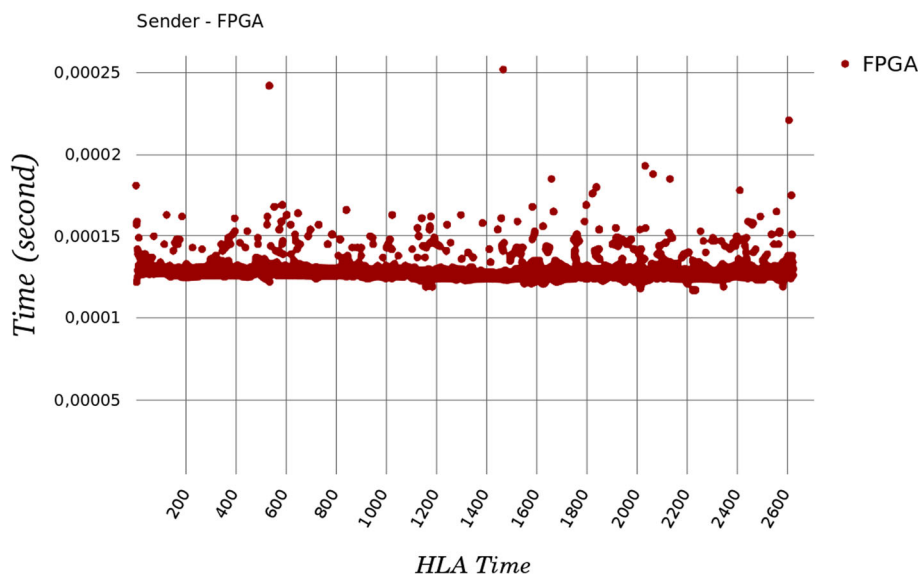


Fig. 11 SoC Federate activity during the communication with Sender Federate

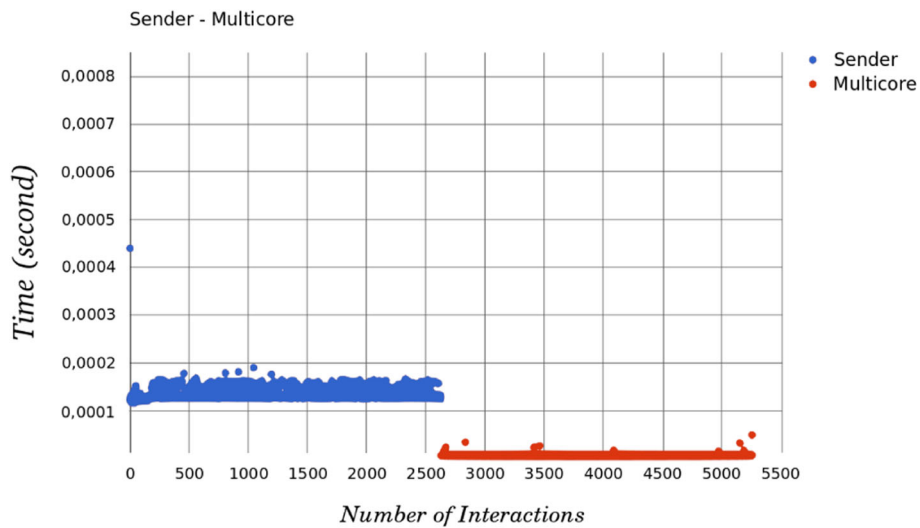


Fig. 12 Transmission activity between Sender and Multi-core Federates

sible to achieve better results while exploring more cores per device.

Table 5 presents the time to process the mask operation over Lena image by the OpenCL kernel in a GPU Federate. For all experiments, the processing time was the same, 281 ms to process the mask, because this time is independent of the transmission strategy. This table demonstrates that the data transfer can be the highest bottleneck in this scenario. Although, the results demonstrated that when transmitting 300 pixels per attribute, the transmission decreases to 45% of overall time.

6.2 Scenario 1: SoC (ARM+FPGA)

The results presented in the following sections show the interaction among the Federates in Virtual Bus. They demonstrate how the data exchanging occurred and when each Federate took action.

In Figs. 10 and 11 are presented the processing activity of the Sender and the SoC Federates, respectively. It is possible to see that both charts have the same shape and the average is $129\mu s$ for each. This is because the SoC returns the result of MD5 hash in the next HLA time after the Sender Federate sends the input data. After receiving

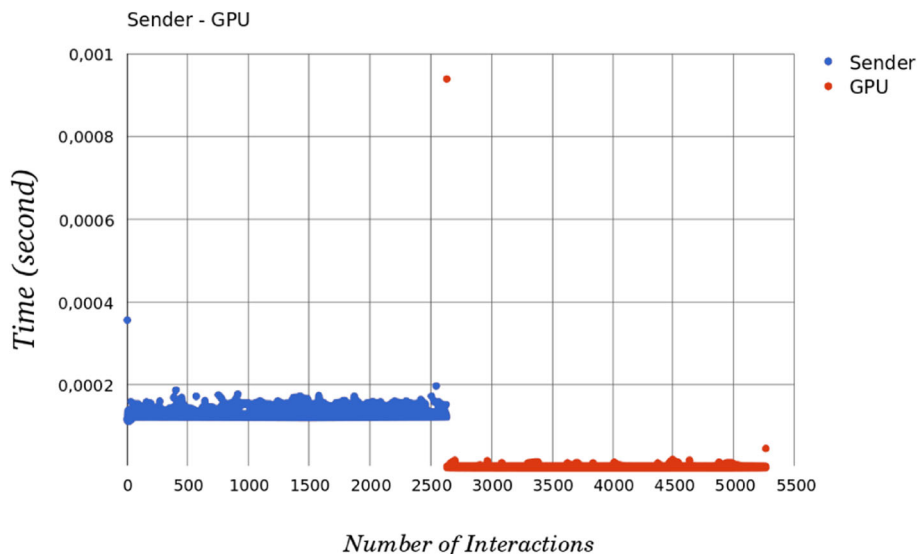
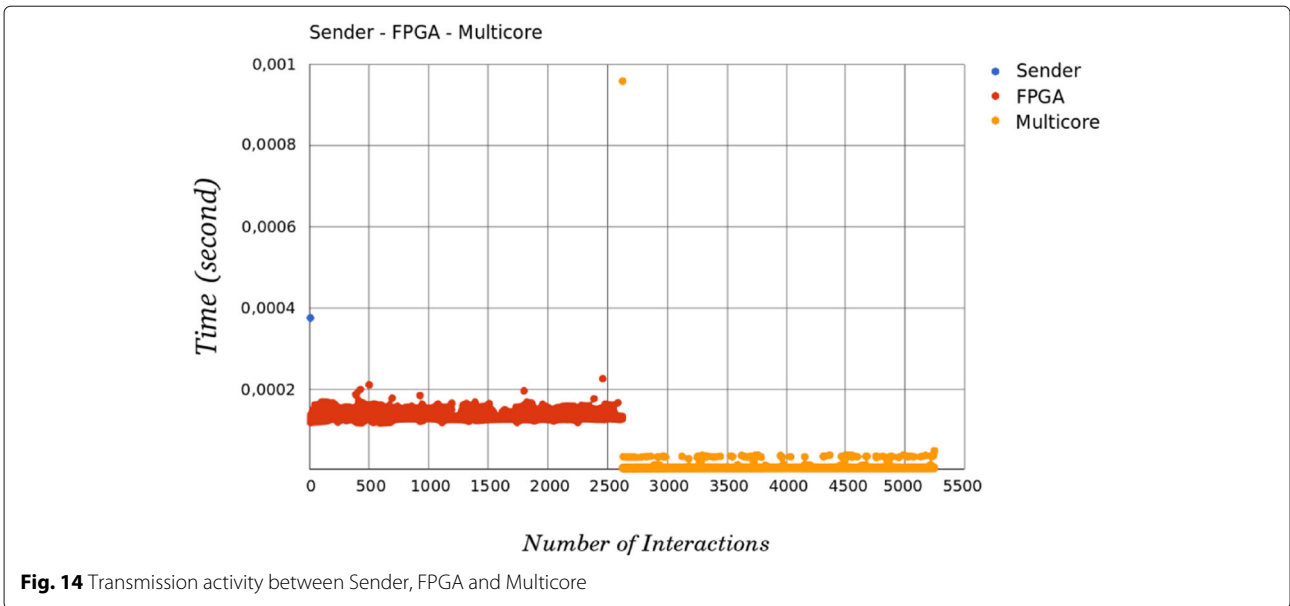


Fig. 13 Transmission activity between Sender and GPU



the last data to be processed, the SoC is the unique Federate transmitting via Virtual Bus, so it expends only $4\mu s$ to conclude the data transfer to Sender Federate.

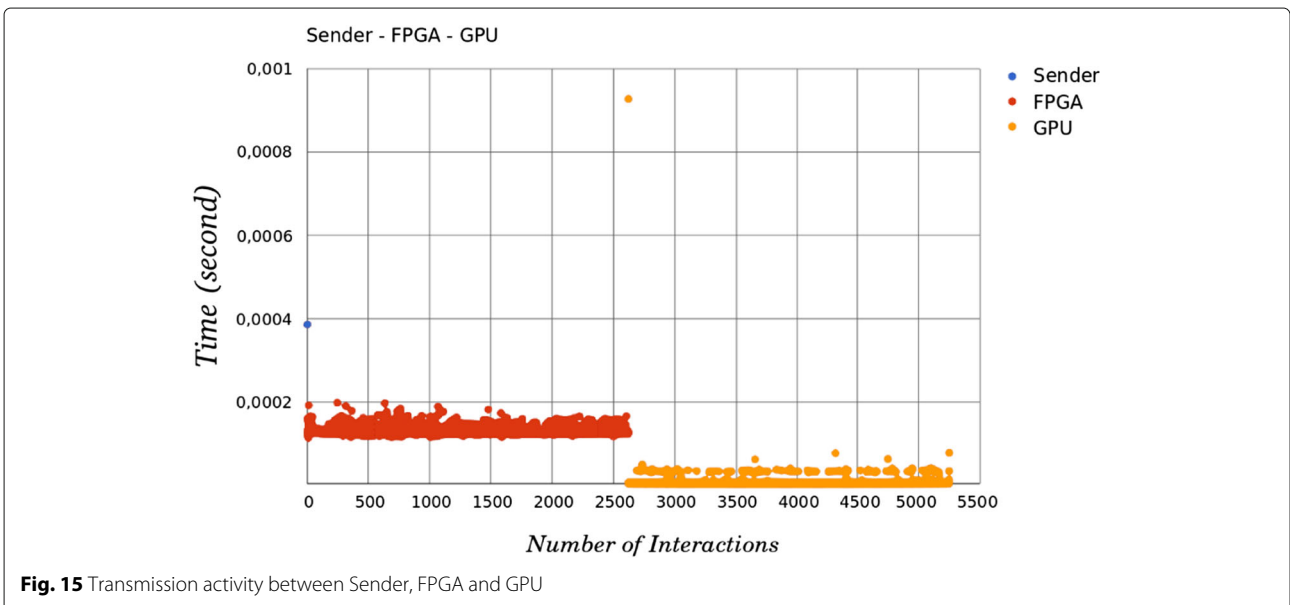
6.3 Scenario 2: Multi-core

Figure 12 presents the communication between Sender and Multi-core Federates. Now the x-axis represents the number of interactions. The Sender sends each message in $131\mu s$ on average to the Multi-core Federate. After receiving all data, the Multi-core Federate takes $811\mu s$ to apply the mask operation and start to send the result to Sender

Federate, this is shown in peak around interaction 2600. The result is sent back in messages, which take $6\mu s$ on average to reach the Sender Federate.

6.4 Scenario 3: GPU

Figure 13 presents the activity during communication among Sender and GPU Federates. The results are similar to the communication between Sender and Multi-core Federates. The only difference is that the GPU takes $940\mu s$ to apply the filter and return the first message to the Sender Federate. Since the focus of this work



is the communication among heterogeneous devices, this code was not optimized for the GPU, resulting in this discrepancy.

6.5 Scenario 4: Multi-core and SoC (ARM+FPGA)

In order to evaluate three devices communicating via Virtual Bus, the Sender, SoC (ARM+FPGA) and Multi-core Federates were connected. The Fig. 14 represents the activity during this communication. The integration of these three devices did not interfere in results that were obtained when two devices were exchanging data. In the Fig. 14 the FPGA activity overlaps the Sender as previously described.

6.6 Scenario 5: SoC (ARM+FPGA) and GPU

The same experiment was made but replacing the Multi-core by GPU, and the result was repeated as could be seen in Fig. 15. A similarity between this result and scenario 4 is clear, since both use the same OpenCL code, and the communication bottleneck continues in same amount.

7 Final considerations

In this work a platform named Virtual Bus for communication between distributed heterogeneous embedded systems was presented. It provides a simple and clear way of exchanging data, without necessity to know in details the architectures involved. The Virtual Bus can also be adapted for many devices, as it is based on the consolidated standard HLA (IEEE 1516).

The experiments demonstrated the communication between different devices using Virtual Bus. Some different devices were integrated in a unique execution environment. A PC, a DE1-SoC with an ARM and an Altera FPGA, a GPU and a Multi-core processor. Once the Virtual Bus was implemented in devices, the communication and synchronization among them were transparent and only three functions were necessary for any application to deal with the bus.

With the experiments it was possible to prove the feasibility of the proposed architecture to perform data transfers preserving consistency of time and content of messages, as well as enabling the necessary infrastructure for parallel processing in a each device connected via HLA. To support massive parallel processing of images, an OpenCL Federate was developed to manage multiple compute units in GPU and multi-core CPU.

The potential and limitations of our platform became evident. The main potential is the possibility to integrate heterogeneous architecture in a transparent and synchronous fashion. The most important limitation is the transmission overhead. HLA is a centralized approach, which is important to manage synchronization but increases the communication bottleneck. However, we have demonstrated that when using array types in HLA

the transmission overhead can be decreased. This possibly enables the use of Virtual Bus by distributed applications which demands synchronization and explore multiple compute units of heterogeneous architectures. For example, multiplayer games, distributed simulation, distributed hardware-in-the-loop simulation, etc. In future works, Virtual Bus will be applied in these and other scenarios. Also, other communication middlewares (e.g. DDS) could replace HLA and the results compared with our current implementation.

Acknowledgements

The acknowledgements goes to the CNPq and CAPES from Brazil for supporting this research.

Authors' contributions

For the development of this work, TWS, AB, AML and EM developed the concept and design of the Virtual Bus. TS, DM and HA implemented the Virtual Bus and the experiments. AML, AB and EM also reviewed the text. All authors read and approved the final manuscript

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

¹Universidade Federal da Paraíba (UFPB), Joao Pessoa, Brazil. ²Universidade Federal de Campina Grande (UFCG), Campina Grande, Brazil.

Received: 28 February 2017 Accepted: 5 December 2017

Published online: 16 January 2018

References

1. Simpson JJ, Dagli CH. System of systems: Power and paradox. In: 2008 IEEE International Conference on System of Systems Engineering. Washington, DC: IEEE Computer Society; 2008. p. 1–5. <https://doi.org/10.1109/SYSOSE.2008.4724165>.
2. IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– framework and rules. IEEE Comput Soc. 2010;2010:1–38. <https://doi.org/10.1109/IEEESTD.2010.5553440>.
3. Scrudder R, Saunders R, Björn Möller KLM. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification. IEEE Comput Soc. 2010;2010. <https://doi.org/10.1109/IEEESTD.2010.5557731>.
4. Brito AV, Negreiros AV, Roth C, Sander O, Becker J. Development and Evaluation of Distributed Simulation of Embedded Systems Using Ptolemy and HLA. In: 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications. Washington, DC: IEEE Computer Society; 2013. p. 189–96. <http://dl.acm.org/citation.cfm?id=2570454.2570892>.
5. Oliveira HFA, Araújo JMR, Brito AV, Melcher EUK. ian approach for power estimation at electronic system level using distributed simulation. *J Integrated Circ Syst*. 2016;11(3):159–67.
6. Wetter M, Haves P. A modular Building Controls Virtual Test Bed for the integration of heterogeneous systems. In: Proceedings of the 3rd SimBuild Conference. Berkeley: Published by authors; 2008. p. 69–76. <http://simulationresearch.lbl.gov/wetter/download/SB08-04-2-Wetter.pdf>.
7. Eidson J, Lee EA, Matic S, Seshia SA, Zou J. Distributed real-time software for cyber-physical systems. *Proc IEEE (special issue on CPS)*. 2012;100(1):45–59.
8. Jung Y, Park J, Petracca M, Carloni LP. netship: A networked virtual platform for large-scale heterogeneous distributed embedded systems. In: Proceedings of the 50th Annual Design Automation Conference. New York: ACM; 2013. p. 1–169:10. <https://doi.org/10.1145/2463209.2488943>.

9. Vaubourg J, Chevrier V, Ciarletta L, Camus B. Co-Simulation of IP Network Models in the Cyber-Physical Systems Context, using a DEVS-based Platform. Research report, Inria Nancy - Grand Est (Villers-lès-Nancy, France) ; Université de Lorraine ; CNRS - Nancy ; Loria & Inria Grand Est. Pasadena: ACM; 2016. <https://hal.archives-ouvertes.fr/hal-01256907/file/paper.pdf>.
10. Van Tran H, Truong TP, Nguyen KT, Huynh HX, Pottier B. A Federated Approach for Simulations in Cyber-Physical Systems. In: Vinh CP, Alagar V, editors. Context-Aware Systems and Applications: 4th International Conference, ICCASA 2015, Vung Tau, Vietnam, November 26-27, 2015, Revised Selected Papers. Cham: Springer; 2016. p. 165–76.
11. Siron P. Design and implementation of a HLA RTI prototype at ONERA. In: 1998 Fall Simulation Interoperability Workshop. Toulouse: Published by authors; 1998.
12. Gervais C, Chaudron JB, Siron P, Leconte R, Saussié D. Real-time distributed aircraft simulation through HLA. In: Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on. Washington, DC: IEEE Computer Society; 2012. p. 251–4.
13. Awais MU, Palensky P, Elsheikh A, Widl E, Matthias S. The high level architecture RTI as a master to the functional mock-up interface components. In: Computing, Networking and Communications (ICNC), 2013 International Conference on; 2013. p. 315–20.
14. Paterson DJ, Hougl ESDP, Sanmiguel JJ. A gateway/middleware hla implementation and the extra services that can be provided to the simulation. In: 2000 Fall Simulation Interoperability Workshop Conference Proceedings, No. 00F-SIW-007. State College: Citeseer; 2000.
15. García-Valls M, Ampuero-Calleja J, Ferreira LL. Integration of Data Distribution Service and Raspberry Pi. In: Au MA, Castiglione A, Choo KR, Palmieri F, Li K-C, editors. Green, Pervasive, and Cloud Computing: 12th International Conference, GPC 2017, Cetara, Italy, May 11–14, 2017, Proceedings. Cham: Springer International Publishing; 2017. p. 490–504.
16. García-Valls M, Calva-Urrego C. Improving service time with a multicore aware middleware. In: Proceedings of the Symposium on Applied Computing, SAC '17. New York: ACM; 2017. p. 1548–53. <https://doi.org/10.1145/3019612.3019741>.
17. Valls MG, Lopez IR, Villar LF. iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *IEEE Trans Ind Inf.* 2013;9(1):228–36. <https://doi.org/10.1109/TII.2012.2198662>.
18. García-Valls M, Cucinotta T, Lu C. Challenges in real-time virtualization and predictable cloud computing. *J Syst Archit Embedded Syst Des.* 2014;60(9):726–40. <https://doi.org/10.1016/j.sysarc.2014.07.004>.
19. Park Y, Min D. Development of hla-dds wrapper api for network-controllable distributed simulation. In: 2013 7th International Conference on Application of Information and Communication Technologies. Washington, DC: IEEE Computer Society; 2013. p. 1–5. <https://doi.org/10.1109/ICAICT.2013.6722799>.
20. Brito AV, Bucher H, Oliveira H, Costa LFS, Sander O, Melcher EUK, Becker J. A distributed simulation platform using hla for complex embedded systems design. In: 2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). Washington, DC: IEEE Computer Society; 2015. p. 195–202.
21. Macri M, Rango AD, Spataro D, D'Ambrosio D, Spataro W. Efficient lava flows simulations with opencl: A preliminary application for civil defence purposes. In: 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC). Washington, DC: IEEE Computer Society; 2015. p. 328–35. <https://doi.org/10.1109/3PGCIC.2015.107>.
22. Weber R, Gothandaraman A, Hinde RJ, Peterson GD. Comparing hardware accelerators in scientific applications: A case study. *IEEE Trans Parallel Distributed Syst.* 2011;23(1):58–68.
23. Noulard E, Rousselot JY, Siron P. Certi, an open source rti, why and how. In: Spring Simulation Interoperability Workshop. Palaiseau: Published by authors; 2009. p. 23–7.
24. Nouman A, Anagnostou A, Taylor SJ. Developing a distributed agent-based and des simulation using portico and repast. In: Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications. Washington, DC: IEEE Computer Society; 2013. p. 97–104.
25. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification. *IEEE Comput Soc.* 2010;2010: 1–378. <https://doi.org/10.1109/IEEESTD.2010.5557728>.
26. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification. *IEEE Comput Soc.* 2010;2010:1–110. <https://doi.org/10.1109/IEEESTD.2010.5557731>.
27. Liu B, Yao Y, Jiang Z, Yan L, Qu Q, Peng S. HLA-Based Parallel Simulation: A Case Study. In: 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation. Washington, DC: IEEE Computer Society; 2012. p. 65–7.
28. Lasnier G, Cardoso J, Siron P, Pagetti C, Derler P. Distributed Simulation of Heterogeneous and Real-Time Systems. In: 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications. Washington, DC: IEEE; 2013. p. 55–62. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6690494>.
29. IETF. Rfc 1321 - the md5 message-digest algorithm. *Internet Engineering Task Force (IETF).* 1992;1992. <https://doi.org/10.17487/RFC1321>.
30. Stone JE, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput Sci Eng.* 2010;12(3):66–72. <https://doi.org/10.1109/MCSE.2010.69>.
31. Cummins C, Petoumenos P, Steuwer M, Leather H. Autotuning OpenCL Workgroup Size for Stencil Patterns, Vol. 8; 2016. 1511.02490.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com