IM - Integration Module for Heterogeneous Devices

Abstract – In this work, an architecture for highly heterogeneous ecosystems is developed, which is integrated in a device called IM (integration module). IM acts as a management and abstraction layer along with the communication and data layer of a service-oriented middleware. The IM at command events in real time by programming groups of threads with the objective to control the abstraction of the heterogeneity of data and the communication protocols of sensors and actuators within a Cyber-Physical System. Part of the IM implementation is presented, and data throughput experiments are performed to measure its performance.

Keywords: Cyber-Physical System, Events, Middleware; Integration - Services; IoT.

I. INTRODUCTION

Cyber-Physical Systems (CPS) explore mechanisms for adapting certain hardware and applications in their capability to integrate with the physical world. The development of a management platform of a sensor network can enable programming, suitability, and design of a Cyber-Physical system. Therefore, it is necessary that the development occurs in an almost intuitive and transparent way, serving as a starting point for the integration of existing autonomous applications.

In [1], some of the advances achieved in the energy aspects of CPSs [2], [3], network security [4], and control and resource allocation in simple applications scenarios are presented to highlight the promising features of CPSs. In [5], a dynamic control middleware (DCM) on the possibility of integrating a CPS with other CPSs is proposed. The proposal is limited to control devices using IPV6. The authors in [5] state that the most appropriate choice would be IPV6, considering the CPS scalability feature. Two types of devices are proposed in the architecture: search control devices and middleware control devices. The proposal includes a mechanism to reduce the exchange of messages generated by a search device. Some other approaches to applications and building of middleware for CPS and device management are described in [6-11].

In the physical world, natural phenomena, as well as the passage of time, are inexorable and the competition between these events is intrinsic. In current designs of various embedded system architectures, these properties should be considered, both in the abstractions of computer processes (applications and operating platforms) and in the architecture of sensor networks. Therefore, software and hardware structures should be carefully designed, considering parameters such as variation, adaptation, efficiency, functionality, abstraction, synchronization, reliability, security, and usability in CPS.

In this paper, it is proposed to design a management layer, named Management Architecture or Integration Module (IM). This layer abstracts the set of events and the processing of transmitted and unstructured data, which are the result of the diversity of hardware offered in the market. In addition, this data available is made available to the CPS using any middleware based on service-oriented architecture capable of adapting to the needs and limitations of actuators and sensor devices.

II. ARCHITECTURE TO MIDDLEWARE

Recent researches revealed some of the challenges in building CPS projects and, among these challenges, there is a need to investigate new middleware projects that would allow to deal with features, such as heterogeneity, component scalability and critical aspects related to sensing, control and application. In this work, it will be presented a middleware based on service-oriented architecture architecture based on middleware services with no further elaboration, since the focus of this work is on the proposed IM, which operates between the set of devices, the middleware and the CPS.

At Figure 1 shows the layered structure of a middleware based on services that integrate the IM layer. The middleware follows a low and high engagement design pattern in some of its layers, centralizing the control of events and pre-processing in the IM layer.
III. INTEGRATION MODULE - IM

The integration module (Fig. 2) is the most important element to control the data flow of devices and the integration with the middleware, given the heterogeneity of the devices operating in the CPS’s control and sensing layer. IM acts as an integration element between sensors, actuators and middleware. Basically, the IM performs the network traffic capture control, executes the insertion and removal of hardware modules, abstraction of routing protocols and interface control. The topology characteristics are answered intuitively by the middleware through the IM, since the organization and routing parameters are transported in the encapsulation protocol in the network and link layer, and these parameters are available in the structure of only one frame (Fig. 2). The IM is integrated into the middleware via the communication and data layer, and can be controlled from the middleware, as an internal component, or from outside the middleware, as a hardware device and external device. Thus, operating over the control logic of a data-sink with a higher level of functionality and abstraction.

The following items describe the IM layers:

- **Interface Layer**: in this layer there are “views”, and can operate directly on the device or can be controlled from the middleware's own interface. Inside of this it, the communication interfaces between the IM and the middleware are managed.

- **Control Layer**: in this layer, the control and scheduling of services and threads for other layers of the IM are managed. The control is realized by pre-defined parameters that come from each of the events, which are the result of the monitored physical phenomena.

- **Service Layer**: in this layer, events can be understood in an easy and intuitive manner. In addition, the following are managed in this layer: data capture services from the MSD (Mote Sensor Device) and MAD (Mote Actuator Device), integration and access to the data and processing layer, and upper layers of control and interface. The services of this layer are designed to be intrinsically interoperable, considering the need to interact with couplings of the various levels between the components of the upper and underlying layers.

- **Processing and Data Layer**: in this layer, data from several MSD and MAD in the CPS are processed. In addition, a data classification method is used based on some of its predetermined variables, such as package size and composition (structured or unstructured data). When a new network connects to the CPS, the middleware should start a process in the communication layer. This occurs so that there is abstraction of the IM interface and the communication protocol in use by the MAD network associated with both IM and MSD. Therefore, the network information set is stored in a buffer so that it can be handled at runtime by the IM processing layer.

IV. EVENTS MODEL - IM - CPS

The CPS largely consists of embedded devices or components, which commonly have feedback loops where physical processes affect predetermined computation, decreasing the system’s reliability. The CPS is considered a distributed hybrid system that, eventually, must handle non-periodic and periodic events with diverse requirements. In [9], the combinations and methods used to treat these events correspond to an optimized real-time CORBA model. The model is optimized in its communication interface to facilitate the configuration and management of non-periodic and periodic events from the underlying layers, which results in increased performance, given the robustness of the CORBA architecture.

Each event managed within the proposed IM associates a set of non-functional spatial-temporal requirements. In the IM, an event is defined as a combination of one or more event conditions, which are restrictions in terms of attributes, time, location, following the model described in [12]:

$$PE_{id}(t_{Eid}, E_{id}, V_{Eid})$$

(1)

Where PEid is the physical event identifier, tEid and Eid are the spatial-temporal occurrences and VEid are the attributes of the physical event. The observed phenomenon is represented as (2):

$$PO(MTid, SRid, i) \{ t_{0}^{i}, t_{0}^{\infty}, V_{O} \}$$

(2)

Where PO is the observed physical phenomenon, SRid is the type of sensor installed on the MTid ( mote id), i is the observed phenomenon, and $t_{0}^{i}(MTid, SRid, i)$, $V_{O}(MTid, SRid, i)$ are occurrences of physical observation in time, place, and attributes, respectively. The sensor event [13] serves as the first level for observations in the model events in the CPS:

$$SE(MTid, Sid, i) \{ t_{S}^{g}, t_{S}^{e}, t_{E}^{g}, t_{E}^{e}, V_{SE}, ps \}$$

(3)

MTid is the mote sensor that generates the event based on the identification of the Sid sensor event at the event instance i.
In addition, a sixth property is used, in which \( t_{SE(MTid, Sid, i)}^{SE} \) and \( l_{SE(MTid, Sid, i)}^{SE} \) are the time and place related to the occurrence of events and mote sensor, respectively; \( t_{SE(MTid, Sid, i)}^{SE} \) and \( l_{SE(MTid, Sid, i)}^{SE} \) are the estimated time of the event, place and attribute occurrences, respectively, according to the mote sensor; and \( p_{S(MTid, Sid, i)}^{SE} \) is the mote sensor degree of reliability in relation to the event sensor.

The set of events is managed by the IM and, at some point, is associated by the CPS through the service layer. These events and sensor attributes are represented in the simplification of (3), presented in (4).

\[
SE(MTid, Sid, i) \{ t_{SE}^{S}, l_{SE}^{S}, v_{SE} \}
\]

(4)

Where \( SE \) is the sensor event, \( MTid \) corresponds to the architecture of the mote, \( Sid \) is a type of sensor over the architecture of the mote, \( i \) is the event observed by the architecture and \( \{ t_{SE}, l_{SE}, v_{SE} \} \) corresponds to the time, space and attributes requirements, respectively.

In the IM, the data classification process is performed by determining common variables in the multihops protocols structure. This set of variables is represented as \( P_M \). For this case, the entrance package data \( DP \) is classified as a multihop package \( DP_M \) when the set of values \( P_M \) is included in its structure (5). The same logic is applied to the singlehop \( DP_S \) (6).

\[
DP = DP_S \leftrightarrow P_S \cap DP
\]

(5)

\[
DP = DP_M \leftrightarrow P_M \cap DP
\]

(6)

It should be noted that many devices transmit unstructured data and this is precisely where IM becomes most efficient. In some hardware architectures, such as Arduino, the processing of the collected event is held locally, transmitting to the IM only the result of this processing. In TmoteSky and Telosb, values of the events are collected and sent to the IM for their pre-processing and temporal storage. These data are usually variable (Figure 4) and will be available to the middleware via the communication and data layer.

Measurement operations are used to quantify some global properties of isolated objects being monitored in the physical environment (field, object size, symmetry, time, etc.) and the relative position of different objects in terms of distance and direction. In IM, spatial analysis operations, which typically use statistical techniques, are used to uncover spatial relationship within and between data layers mapped from the cyber-physical environment scenario, such as object location and control of events inside a smart house.

V. IM STRUCTURE DEVELOPMENT

In IM, the interfaces control with other IMs and middleware, data flow and integration with the middleware are performed using thread packages. These packages contain the definition of operations and conditions to create new threads. They also contain aspects related to synchronization variables and mutual exclusion of condition variables to maintain the cycle of an abstraction and data processing operation between the MSD and the IM or another IMs.

When the MSD and MAD are connected to the IM, the IM waits for an entrance request for a data flow operation transmitted by the MSD, and then, the IM performs the request and returns the response. In this case, a thread reads requests that arrive for a data flow operation that needs to be structured, and the requests are forwarded by the IM using an already known port. After examining the request, the control module chooses a second thread, yet blocked, and delivers the request to it. The first thread remains suspended until the next input of MSD data. When the thread is linked to an IM operation (a specific data flow type of an MSD or MAD), another thread is selected to execute requests from other networked MSD architectures, maintaining the IM heterogeneity supporting logic. Therefore, there will be different mini-processes or execution flows of the same IM connecting to the middleware storage module and responding to the various MSD and MAD architectures (Fig. 3).

![IM connection structure](image)

Fig. 3: IM – connection structure.

At Figure 3 shows more than one IM integrated with the middleware to manage the high heterogeneity of devices and data resulting from the communication between MSD and MAD in the CPS. When necessary, it is considered the possibility of including an MSD that acts as a data sink and leads the network data to the IM for the abstraction of communication protocols, topologies and communication patterns existing in the communication environment.

The initial option in development is to control the IM from the middleware interface using a start method (start button), which initiates the "startcapture" method. In the middleware interface, the "start capture" method makes a "request" for the control level represented in the "CaptureBean" class, as represented by the code following:

```java
<h:outputLabel value="Port Name:"/>
<p:inputText value="#{captureBean.port}"/>
<h:outputLabel value="Port Speed:"/>
<p:inputText value="#{captureBean.speedPort}"/>
<p:commandButton value="Start Capture" action="#{captureBean.startCapture}" ajax="false"/>
```

The "startCapture" method of the "CaptureBean" control class is instantiated in the "CaptureSensor" class and, there, the call of the "captureConfig.getCaptureSensor().Start();" method is initiated, in which the capture is initiated through the IM, as represented by the code following:

```java
public void startCapture() {
    try {
        // Code to initiate capture
    }

    catch (Exception e) {
        // Handle exception
    }
}
```

The possible use of condition variables to maintain the cycle of an abstraction and data processing operation between the MSD and the IM or another IMs.
In the "CaptureSensor" class, the "service facade" (facade pattern) is declared, which enables the instance creation in the constructor, selecting the desired implementation.

```java
private final SensorService sensorService;

public CaptureSensor(String port, int speedPort) {
    this.port = port;
    this.speedPort = speedPort;
    sensorService = new SensorServiceImplByORM();
}
```

Still in the "CaptureSensor" class, there is the implementation of the "serialEvent" method, which is an implementation of the "SerialPortEventListener" interface. This method is added as a "listener" and it is performed with each action on the IM port, calling the "Capture" method and sending a clone of the "buffer" as a parameter. At this point, the data captured from an MSD is ready to be processed at run time, while the first thread is suspended and a second thread continues the capture process. Represented by the code following:

```java
@Override
public void serialEvent(SerialPortEvent spe) {
    try {
        Thread.sleep(60);
        int a = inputStream.available();
        byte[] readBuffer = new byte[a];
        while (inputStream.available() > 0) {
            inputStream.read(readBuffer);
        }

        sensorService.Capture(readBuffer.clone());

    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    getSerialPort().addEventListener(this);
}
```

In the capture method of the "SensorServiceImplByORM" class, the "CaptureProcess" class is instantiated and it is called the "start ()" method, which is the class inherited from the main thread. Consequently, a new thread is started to process the package.

```java
@Override
public void Capture(byte[] captureData) {
    new CaptureProcess(captureData).start();
}
```

In the "CaptureProcess" class, all DAO are started to perform the persistence tasks, such as querying, writing, and deleting data in the database shared between the IM and the data module of the middleware communication layer. Represented by the code following:

```java
private DaoORM<Sensor> daoSensor;
private DaoORM<PackageTmoteMultihop> daoPackageTmoteMultihop;
private DaoORM<PackageTmotePeerToPeer> daoPackageTmotePeerToPeer;
private DaoORM<PayloadTmoteMultihop> daoPayloadTmoteMultihop;
private DaoORM<PayloadTmotePeerToPeer> daoPayloadTmotePeerToPeer;
```

Considering that the implementation described corresponds to the MI control from the Middleware, the information abstracted from the network through the MI module is stored in a structured way in the existing database on a remote architecture. The MI establishes a communication with the data module through a "run" method, which is an overridden method of the "thread class", it runs every time a thread is started with "start ()". In the "run" method, packages are processed and persisted in the bank. In a script, the points are described where iterations between the MI data module and the data module of middleware communication layer are identified.

Some of the classes, such as "PackageTmoteMultihop" and "PackageTmotePeerToPeer", are strongly coupled and prevent an easy change or update of their components and services structure. This is important, as parts of the structure of the communication protocols used as singlehop and multihop have unique characteristics in their structure, and stability depends on how services extract and store routing information and data. This strong coupling characteristic is necessary to prevent the user or the CPS administrator from attempting to extract an excessive amount of information, hindering the performance of the application at run time.

VI. PERFORMANCE AND EVALUATION SCENARIO

To evaluate the IM’s architecture performance, the raw speed and the number of MSD give a good idea of the progress that has been achieved. However, to evaluate the performance of computing systems, more precise metrics related to the execution of programs are needed. From the point of view of the CPS user, speed and transparency with which the events are performed are the most important factors. However, this execution time depends on many factors that have little to do with the architecture of the processor on which the programs run. The running time of the IM will depend on the number of events executed and the time needed to execute each of them (7).

\[ T_{pse} = NSE \times TPI \]  \tag{7}

Where \( T_{pse} \) is the time to process a pse (processing sensor event), \( NSE \) is the number of sensor events and \( TPI \) is the time to process an instruction.

Three hardware architectures are considered in the test scenario: the first is the TmoteSky, the second is the XBeePRO, and the third is the Arduino Duemilanove. For all architectures, a temperature sensor was coupled.
For the Arduino Duemilanove, a Shield with a XBeePRO module was coupled to achieve the RF transmission along with the other architectures mentioned. Therefore, the temperature processing of the event was programmed locally, given the increased computational power offered. Part of the temperature sensor coupled code is shown, as follows:

```c
float tempC = sensor.getTempC(deviceAddress);

void printTemperature(DeviceAddress deviceAddress) {
    Serial.print("Temp C:");
    Serial.print(tempC);
    Serial.print("Temp F:");
}
```

All motes transmit temperature data collected in a home scenario. Values are expressed in degrees Celsius and Fahrenheit. In the case of Tmotesky and XBeePRO motes, the data is transported and collected by the IM, according to the structure shown in Figure 4.

The set of tests performed between the MSD and the IM changes according to the number of packets transmitted by varying the sampling rate (time rate). In Figure 5, 500 packages at a time rate of 5 ms are transmitted between the MSD and the IM. The first test shows a package loss of up to 2%; however, it should be noted that the set of events managed by the CPS is of a periodic and non-periodic characteristic and not continuous. Additionally, no matter what the event is, a high sampling rate would not be required, considering the density of devices in the implementation of the CPS.

In Figure 6, the results achieved at a time rate of 250 ms are presented. It was also considered a larger number of transmitted packages (1,000 packages), and it can be seen that the loss is zero.

In Figure 7, the same test is performed. However, the number of samples is changed to 2,000 and the time rate, to 1,000 ms. These values would be acceptable for periodic samples of events for most sensors applications.

In the tests execution, a time tag was inserted along with an ID for each frame transmitted in order to maximize the fidelity of the tests performed. This was done due to the homogeneity of the devices and the multihop and singlehop protocols characteristics in which they were tested.

VII. CONCLUSIONS

A variety of hardware architectures and communication protocols are used in a CPS, so services in the communication layer must initiate and implement methods and components that abstract this diversity of protocols and topologies, creating a common environment to all of these elements. When a new network and the set of MSD and MDA connect to the CPS, the middleware should start the IM at the application interface and the abstraction process of the connection should interface with the IM at the communication layer. This includes the communication protocol (multihop or singlehop) in use by the MSD and stores the set of MSD network information in a buffer so that it can be handled at runtime by the IM or the storage module in the middleware communication and data layer.

The IM abstracts information from the network and from the MSD and saves the package information in the storage module (processed). The MSD package provides routing and sampling of events (physical phenomena), which triggers
other MAD transducers and systems integrated to the CPS. Some of this information is treated at runtime and made available to the CPS user and others are kept in the storage module in case the administrator or the CPS user needs it.

Each MSD and MAD network has different data structures due to the heterogeneity of communication standards and protocols used. In IM, it is possible to have a data sink for each communication standard used in the CPS environment connected between the MSD and the IM, facilitating IM’s performance and bringing a stable behavior of the system, where everything is handled in the application layer.

REFERENCES