CAP: Context-Aware Programming for Cyber Physical Systems

Shashank Gaur
CISTER Research Centre
ISEP, Polytechnic Institute of Porto
Porto, Portugal
sgaur@isep.ipp.pt

Luis Almeida
CISTER Research Centre
FEUP, University of Porto
Porto, Portugal
lda@fe.up.pt

Eduardo Tovar, Radha Reddy
CISTER Research Centre
ISEP, Polytechnic Institute of Porto
Porto, Portugal
emt, reddy@isep.ipp.pt

***Abstract*—Context-awareness is a prominently desired feature in computing systems. Smartphones, smart cards or tags, wearables, sensor nodes, and many other devices enable a system to compute context for different users and environment. With ever increasing advances in hardware for such devices, the interactions with users are increasing every day. This enables the collection of a large amount of data about users, systems, and physical environment. With such data available to be leveraged, context-awareness will soon become a necessity. Such type of data collection happens most frequently in sensing applications enabled by wireless sensor network (WSN) devices. This paper discusses the concept of context for sensing applications, specifically related to Cyber Physical Systems (CPS). The paper highlights key aspects of context and its definition. This paper proposes, to the best of the author's knowledge, the first programming approach to build context-aware applications for WSN-based CPS. This paper provides a proof of concept for a framework to detect, manage and deploy context-aware applications.**

***Index Terms*—Cyber Physical System, Wireless Sensor Network, Context Awareness, Dynamic Reconfiguration, Adaptability, Middleware**

I. Introduction

Sensors are used to monitor events in their environment for various applications such as healthcare, structural monitoring, data centers, agriculture, process and production industry etc. A network of such devices, commonly known as Sensor Network, is able to collect data and exchange processed information with other devices to provide new services to the user. With evolution in technology, the devices in sensor network became more advanced to collect data, communicate over wireless network, and process the data into useful information. This has significantly changed the way computation and communication of the data happens, particularly, in the so-called Wireless Sensor Networks (WSN)s. WSN devices with sufficient processing and storage capabilities enable the user to perform sensing related tasks over an extensive period of time. In addition, WSN also enable different interactions between the environment and the user.

Such WSN devices are prominently used in Cyber Physical Systems, specially in Industrial CPS. For example, in a warehouse smart RFID tags are used to keep track of the movement of an object. Motion sensors in a production plant can allow efficient HVAC operations across occupied areas in real time. Such complex use cases with WSN-bsed CPS already exist [1]. In the recent years, there has been efforts to enable efficient use of WSNs in industrial CPS domains such as Manufacturing Automation, Process Automation, Production Management, etc.

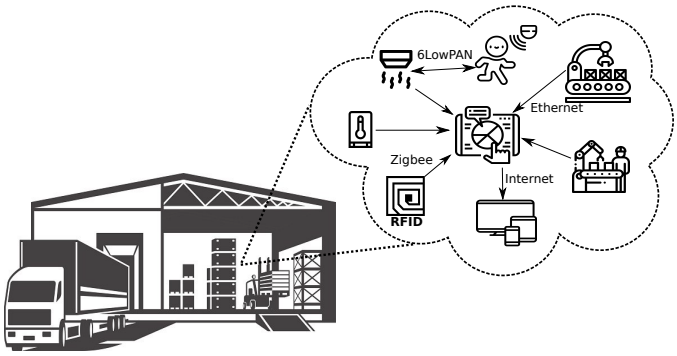

Fig. 1: WSN in Industrial CPS

Due to the broad potential, there has been tremendous progress to enable these WSN-based CPS. On the hardware front, the devices are powerful enough to execute and process multiple tasks with help of various onboard sensors. Protocols such as Zigbee and 6LoWPAN allow devices to communicate without the Internet [2], [3]. On the software front, many solutions are available for users to write applications [4], [5].

Such progress also yields an increase in expectations from the industry. An easier way to program the applications and better interactions with other devices are desired. Several research works have been carried out towards satisfying such expectations in various scenarios [6], [7].

On the other hand, Cyber Physical Systems have become much more pervasive and ubiquitous. A single device can collect different type of sensor data such as location, temperature, motion, etc. Such semantic expansion of data allows for a better understanding of the ecosystem. It is possible to deduce various contexts and also adapt based on such deductions. This is generally known as Situational-Aware or Context-Aware Computing.

This paper discusses how context-awareness can be enabled in CPS, especially with help of WSN devices. The paper also proposes novel programming solutions and operational framework, which allows the user to leverage such property.

978-1-7281-0303-7/19/$31.00 ©2019 IEEE

This work examines the requirements and essential design features for such programming solutions.

This paper is organized as follows. Section II describes the concept of context awareness and its relevance. Section III discusses possible programming solutions and their features to enable context awareness. Section IV presents the implementation of the proposed programming solution. Section V evaluates the implementation and examines the benefits of those proposed features. Section VI discusses relevant work and Section VII provides a conclusion and future direction for this work.

II. Context-Awareness

Context awareness [8] can help in building a true pervasive Cyber Physical System. It can anticipate the needs of a user/environment and act accordingly. This is an effort to free everyday users from manually configuring a system for all possible needs. A simple example of context-awareness can be found in the Cyber Physical Production Systems [9]. In production systems context can change depending on the type of product, time, temperature, etc. For example, a cooling liquid is used to control temperature of manufacturing machines, which is supplied based on the temperature detected by the sensors. However, some adaptation is required based on the time taken by cooling process and change in the temperature. There can be complex scenarios where type of the product being manufactured may also affect the cooling process. Such a cooling application is using a wide variety of contexts to provide this service.

Context itself has multiple definitions in different use cases [10]–[12]. The definition by Abowd *et al.* [13] is the closest to a desired operational definition for this work. The context can include information related to three aspects of the ecosystem:

- ***User*** such as location, motion, nearby users, etc.
- ***System*** such as applications, connectivity, energy, nearby devices, etc.
- ***Physical Environment*** temperature, time, noise, etc.

With help of information about these three aspects, *Context* can be defined. Especially in the case of WSNs, precise data extracted from sensor devices can help in defining the context for the system.

Definition: *Context is the information that can be used to characterize the situation of the ecosystem. A Context can be determined based on information from a person, place or object that is relevant to the users and their applications.*

Once a context is determined for that system, it is possible to execute certain applications which satisfy the user needs for that particular context. Such context-awareness can help in anticipating the changes in the ecosystem and adapt itself accordingly.

To better understand the need for Context-Awareness, let us consider a manufacturing line at a production plan using multiple robotic arms. Both arms are capable of performing operations for various products. From time to time, the arms may need to perform operations in series or parallel, as shown in the figure below.

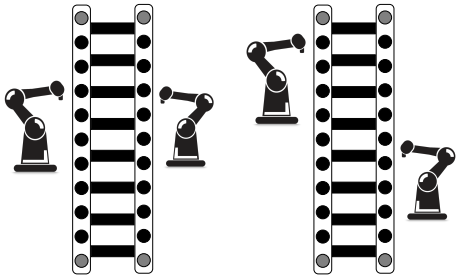

Fig. 2: Use case for context-awareness

The operations performed by these arms may require distinct configurations for multiple contexts. One type of product may require both arms to work together while another may require one arm to perform operation before another arm. In another instance, only one arm may be required for a certain operation. Also time of the day may play certain role, as during certain hours required productivity might be higher. All these configurations can be provided by the user in advance and as the context changes, the arms may adapt by itself.

This use case is used just to introduce the intuition of context-awareness, but the basics for context can exist in many other foreseeable scenarios. Some relevant research work for the similar application already exists [14].

There can be multiple ways to provide context-awareness. In principle, the system needs to determine context with changes occurring in the user, system or physical environment. Once the context is determined, the system must decide on which applications to execute for a particular context. Once the set of desired application is identified, these applications must be deployed across various mobile devices and sensing nodes.

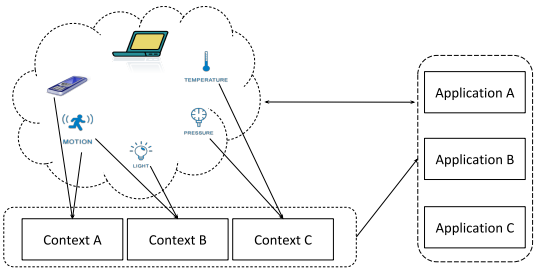

Fig. 3: Adapt applications for different contexts

To determine the context, the system needs to collect the data from all the devices and perform regular computation on the collected data. In the case of WSN devices, this computation can be a set of simple rules on the sensing data. More complex methods, such as machine learning, may be required to recognize more detailed information about user, system or physical environment. For example, information about the social activity of a user may require complex

methodologies to determine such context. However, this work concerns mostly with WSN devices. The context-awareness in this work is achieved using a set of specific rules applied to the data collected from WSN devices.

III. Context-Aware Programming (CAP)

To support context-awareness, it is important to provide the user with appropriate tools to write the applications. There have been many efforts to provide better programming support for users [15], including some support for adaptation in WSN [7]. The main goal of such programming approach should be to let the user express the desired goals without requiring knowledge about specific resources. Resource-agnostic programming and mobility of applications over resources can support context-awareness.

In addition to enhancing the programming, it is important to develop a better understanding of the *context*. This can be achieved by defining the *context* for different scenarios related to security, energy, communication, user behavior, etc.. For any system, these context scenarios can change with time or multiple scenarios can exist at the same time. To adapt to new context scenarios the system must take actions. These actions can be the deployment of new applications across the network or re-configuration of an existing application for a different resource.

This paper proposes a declarative approach to program user applications, named Context-Aware Programming (CAP). In this approach, a user can write self-contained blocks of code, which can process a predefined type of input and provide a certain type of output. We draw upon three essential features from existing state of the art and propose an approach to bring these together.

- *Abstraction* allows the user to write code without worrying about low-level details.
- *Modularity* allows the user to write code that is reusable in modules to provide specific functions.
- *Mobility* allows the user to write code which can be moved around the network across any suitable device.

To understand these features in a better way, let us take a look at a simple example for HVAC operation shown in Figure 4. *Application A* takes input from the two temperature sensors and provides the average temperature as output. *Application B* checks for user presence using the input from a motion sensor and the average temperature from *Application A*, and if conditions are satisfied actuation for appropriate heating or cooling takes place. *Application A* is deployed on one of the temperature sensors and *Application B* is deployed on the actuation device for the HVAC. There are two motion sensors available which can be used by *Application B*.

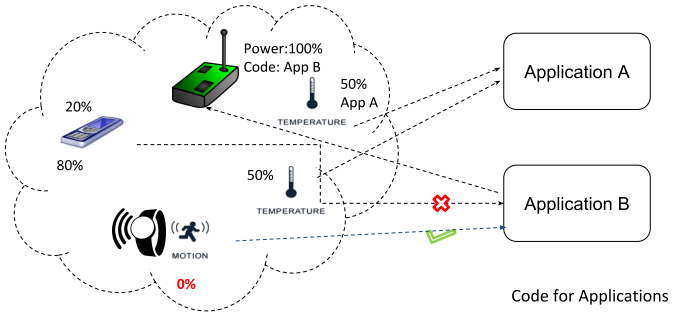

Fig. 4: Features for Context-Aware Programming

Assume the motion sensor provided by the wearable device becomes unavailable in the network, either due to low energy levels or connectivity. *Application B* can still use another motion sensor available via the smartphone. In order to accommodate this change, the code for *Application B* must not be bound to the hardware address of one motion sensor. The user should be able to write the required input without specifying each device to be used, but just the input data required. This is where *Abstraction* feature is required for CAP. Similarly, the complete HVAC application to detect user presence and calculate an average temperature could be written all together, and usually would be done in that way using traditional WSN programming tools such as Contiki. However, dividing this objective into two self-contained applications provides the ability to change the input sensors for one application of the whole objective without interrupting another application. *Modularity* helps in providing such ease of access for the system. In another scenario, one of the temperature sensors hosting *Application A* may become unavailable in the network. In that case, *Mobility* allows the system to deploy *Application A* on the other device such as the smartphone to get the temperature as an input.

Abstraction, *Modularity*, and *Mobility*, altogether enable the user to write applications for different contexts. When a context change occurs, the system can make appropriate adaptations without any manual configurations by the user.

IV. Implementation

In order to implement CAP, this work takes inspiration from some of the existing implementations for programming support in WSN. This work is an adaptation of such implementations towards achieving a solution to accommodate context-awareness in mobile sensing systems. One of such prime inspirations for this work is T-Res [7].

T-Res is an abstraction for programming applications, specifically designed for the Internet of Things. In T-Res, each application (i.e. *T-Res Task*) consists of four components as following:

- *Input Source (/is)* collects input data from other devices for the *T-Res Task*.
- *Processing Function (/pf)* is the code to process input for the *T-Res Task* itself.
- *Output Destinations (/od)* is the destination devices where output of *T-Res Task* is posted.

- *Last Output (/lo)* stores the most recent output generated by *T-Res Task*.

IPV6 based URI addresses are used to assign Input and outputs to */is* and */od*. A compiled file *T-Res Task* is provided to */pf*. T-Res uses CoAP operations such as Put, Get, Post and Observe to perform these actions.

TABLE I: T-Res task with its components and CoAP methods

Tasks	Sub-resources	Possible actions	CoAP methods
Task_Name	input sources (/is)	fetch/update	GET, PUT, POST
	processing function (/pf)	fetch/update	GET, PUT
	output destinations (/od)	fetch/update	GET, PUT, POST
	last output (/lo)	fetch/observe	GET

T-Res is able to keep input and output parameters separate from the application code. This is an extremely useful feature, which is used to implement CAP as well. T-Res has similar abstraction between the code written by the user and devices to be used for the code. Another example of similar separation is PyFuns [16], which also inspired the implementation of CAP.

CAP strives to provide the three features altogether for context-awareness: *Abstraction*, *Modularity* and *Mobility*. CAP is divided into three components as shown in Figure 5, *Application Manager*, *Resource Administrator* and *Context Manager*. The *Context Manager* collects information on the context and compiles a list of applications associated with each context. The *Application Manager* collects code from the user for each application and also keeps track of active applications. The *Resource Administrator* deploys the code to host devices, assigns the input and output devices and keeps track of any changes in the system.

CAP is implemented using Python and Django programming languages. The code for applications is written in *nesc* by the user, similar to T-Res and many other popular programming solutions. Due to its popularity, *nesc* allows familiar users to use CAP without learning another new programming language or syntax. CAP utilizes multiple Python scripts to iterate through the code provided by the user. CAP adds additional flags to the *nesc* code and compiles the binary file to be deployed on the devices using CoAP operations. With help of these flags across the code, whenever there is a context change, the code can be recompiled and deployed again for the appropriate resource. However, these actions are performed autonomously by CAP to assure continuous execution of the applications, without intervention from the user. CAP utilizes an open source library *txthings* [17], to provide support for the CoAP operations in Python.

A. Context Manager

The *Context Manager* is enabled by a set of Python scripts named *context-dict*, *context-input* and *context-track*. These scripts altogether keep track of every context and the applications associated with each context. For an application to be associated with a context, it must have been already created using the Application Manager. A context is created using dictionary data type in Python. Each context has four keys:

- *id* is used for identification using an integer value.
- *group* denotes the ranking of individual context, which is used to resolve conflicts between multiple contexts.
- *applications* contains a list of all associated applications.
- *triggers* is a list of items which can affect the context, such as output of devices, application, or user inputs. A separate list is created with the conditions for each of these triggers.

With help of these, the scripts are able to detect and inform other components about a change in context.

B. Application Manager

The *Application Manager* utilizes a web form to collect code from the user for each task.This form is built using Django and has a backend in Python to refresh the list of applications. This form has four input fields: input, output, host and code.

- *Input* is the type of input required for the application. User can perform an abstract selection from all the available resources using a dropdown list. There is no need to provide an address or details of a particular device, just the type of resource is required. For example, if there are two temperature sensors, three motion sensors, and one pressure sensor available for the user, the web form will show the user a drop down menu with just three choices: temperature; motion and pressure.
- *Output* is the output destination for the application. It is similar to the *Input* field.
- *Code* is the nesc code written in an abstraction similar to T-Res.
- *Host* is the devices where the *Code* will be executed. It is similar to the *Input* and *Output* field.

These four fields are enough to complete an independent application. Sometimes more information about the devices may be required such as time bounds, spatial limits, etc.

C. Resource Administrator

The *Resource Administrator* is also enabled via Python scripts, which also provide automated CoAP operations (such as PULL, PUSH, GET, and OBSERVE). As shown in Figure 5, *Resource Administrator* creates and regularly updates two tables. First table is about status of all the devices available and resources provided by those devices.[1]

After the user submits an application via the provided web form by the *Application Manager*, the *Resource Administrator* takes decisions on which resources to use to host the code, take input and provide the output. It creates a dictionary with the list of acceptable resources for each decision, using the first table. If any of these lists are not created or are empty, the framework notifies users that the desired resources are not available. Once the decision is finalized, the *Resource Administrator* creates a second table with these decisions, as shown in Figure 5.

[1]From here on *resource* defines "sensing resource" per device, e.g. a smartphone has multiple *resources* such as temperature, location, motion, etc.

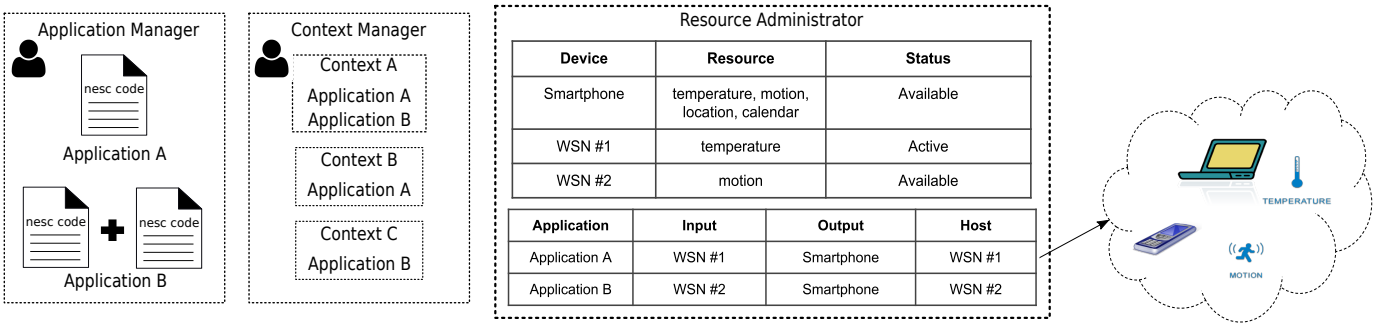

Fig. 5: Components of Context-Aware Programming

Every time any of the tables is updated, the *Resource Administrator* will execute the CoAP operations to deploy the code on the selected host device and assign the input and output devices using the URI addresses. For this it will choose one of the options from each list with a predefined-criteria and keep the selection saved for future references.

```
hostdevice = resource[host_type][available]
inputdevice = resource[input_type][available]
outputdevice = resource[output_type][available]
assign(inputuri, inputdevice)
assign(outputuri, outputdevice)
assign(codeuri, open(code).read())
post(taskuri, "Start")
```

Listing 1: CoAP operations to assign resources.

The CoAP operations such as PUT and POST are executed to deploy the selections and run the task, respectively. The GET operation is executed to make sure that selections are made correctly, as it returns the status of each resource. This can be seen in Listing 1, where *resource* is the dictionary with lists of all acceptable resources. The *host*, *input* and *output* are selected from this dictionary. The functions *assign* and *post* refer to the CoAP requests for PUT and POST respectively. The status of every assigned resource is changed from *available* to *active*.

```
host_status=checkresource(hosturi, hostdevice)
input_status=checkresource(inputuri, inputdevice)
output_status=checkresource(outputuri, outputdevice)
if host_status!="Active" or input_status!="Active"
        or output_status!="Active":
        application.restart()
```

Listing 2: Check status of resources.

As the application is deployed and execution starts, the *Resource Administrator* updates the status of all active resources regularly. This is done by performing GET operations via the Python function *checkresource*, as shown in Listing 2. At any point, if any operation returns with an error, the framework will once again execute the process to allocate resources. However, this time it will use another available resource for the corresponding error received earlier.

V. Working Demonstration

This work was tested using the TinyOS [5] and TelosB sensor nodes. For the demonstration, a simple HVAC system as an example is considered. There are four TelosB nodes as shown in Figure 6. The functions of each node is as follows:

- Mote 1 acts as a border router node
- Mote 2 acts as a host temperature sensor node
- Mote 3 acts as a input temperature sensor node
- Mote 4 acts as an output heating actuator node

Both sensor nodes 2 and 3 can measure the same physical parameter. The host sensor node 2 takes input from the sensor node 3, divides the input values in half and provides output to the actuator node 4.

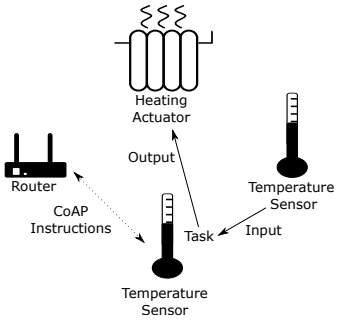

Fig. 6: Four nodes with a simple HVAC application

In T-Res, these three devices have to be connected by a PUT request of CoAP. The compiled code of the task is also deployed using another PUT request to the uri path of host node 2. To complete the deployment, a POST request to host node 2 is required. In T-Res the user is required to issue all these CoAP requests via the Copper CoAP [18] user agent for Firefox. In CAP, the user can provide the same code using the application form provided by the *Application Manager*. The *Resource Administrator* takes care of all CoAP operations.

TABLE II: Status table during operation

Application	Halve Task	Running
Code	halve.c	Sent
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	coap://[aaaa::200:0:0:3]/sensor	Active
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

As soon as user submits the application, the CAP will return the user with a success message for the deployment and a table with status of nodes being used for the current deployment II. This table is refreshed regularly via performing GET requests in the backend. A time bound for those request can also be provided by the user. User may also force a refresh via options provided by CAP.

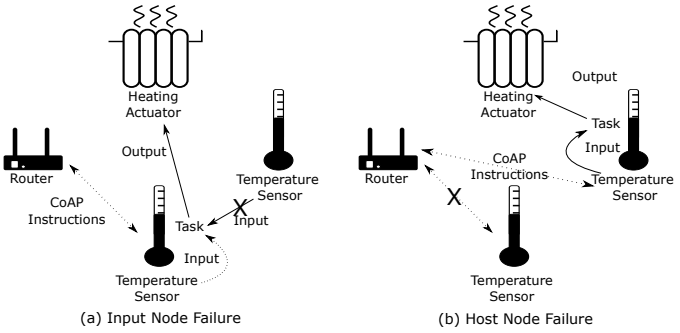

Fig. 7: Four nodes with a simple HVAC application

A. Change in Context

In this example we take a look at a change in context due to energy failure. it is demonstrated the actions of CAP on failure of two nodes, host and input, respectively as shown in Figure 7. First, let us assume that after some time of operation, input sensor node 3 fails due to battery depletion, causing a switch in context as shown in Table III. The CAP will automatically reinitialize the deployment by substituting the node 3 with node 2. The *Resource Administrator* performs PUT request for input source as node 2. After this, normal execution of the application resumes. This new status is represented in Table IV. The sequence of operations is shown in the diagram in Figure 8.

TABLE III: Status table with failure status

Application	Halve Task	**Halted**
Code	halve.c	Running
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	**coap://[aaaa::200:0:0:3]/sensor**	**Inactive**
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

In a second case, host sensor node 2 may fail instead of the input sensor node 3. In that case, the CAP will reinitialize the deployment by substituting the node 2 with node 3 as host node and assign itself as the input source as well. Once again, this would be done by *Resource Administrator*, which performs two PUT requests for both code and input respectively.

TABLE IV: Status table after redeployment

Application	Halve Task	**Running**
Code	halve.c	Sent
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	**coap://[aaaa::200:0:0:2]/sensor**	**Active**
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

As we see in above example the user is not required to intervene as the system was able to detect the context change and adapt to that. Our implementation builds on top of the support provided by T-Res, and extends it to support context-awareness. Many features of the extension can be further developed to enhance the context-awareness. For example the time-bound for refresh on status table can be pre-defined by the user or can be deduced by algorithms for better resource management.

VI. Related Work

Building programming abstraction for WSNs is a major research direction. There has been significant contribution towards macroprogramming, to name a few of them are Regiment [19], Abstract Task Graph [20], Profun [21] and Pyot [22]. Some of these efforts are limited to WSN devices while others try to support mobile devices as well.

Regiment is a functional reactive programming model, which treats the outputs of sensor nodes as Streams [19]. A programmer can write functionalities based on these streams instead of worrying about the nodes. There are some basic functions such as *rmap*, *rfilter*, and *rfold* to operate on these streams. The streams can be combined into groups which are called regions. Due to a functional approach, the regiment provides a high level of accuracy in performance.

While Regiment is a Functional approach, Abstract Task Graph (ATaG) is a data driven approach. In ATaG, every application is divided into three declarations: *Abstract Tasks*, *Abstract Data* and *Abstract Channels*. Abstract Tasks represent the type of processing in any application, Abstract Data represents the type of data handled by the applications and Abstract Channel associates the task declaration with data declaration. Using these declarations any application can be described by a model and then that model can be instantiated any number of times throughout the sensor network. ATaG provides abstraction because the number and the placement of the application can be determined at compile or run time according to the target devices.

The above mentioned efforts do not take into account the diversity of the devices included in possible scenarios. Increase in the mobile devices bring in a number of new challenges such as in network processing, heterogeneity of software and hardware platforms, resource management, etc. There are other recent contributions that try to contribute towards solving these challenges. Nano-CF [23] is another framework which supports in-network processing and concurrent application that share same resources. It batches multiple applications together to optimize the resource and network usage.

Python-based framework (PyoT) [22], aims for the coordination of activities of a group of IoT agents based on CoAP. These are namely discovery, monitoring, and storage. Pyot also aims to coordinate triggering of devices and their data. PyoT focused on sensing and actuation by hiding communication details as objects and supports in-network application processing through T-Res. For task distributions, a message-oriented middleware (Advanced Message Queuing Protocol

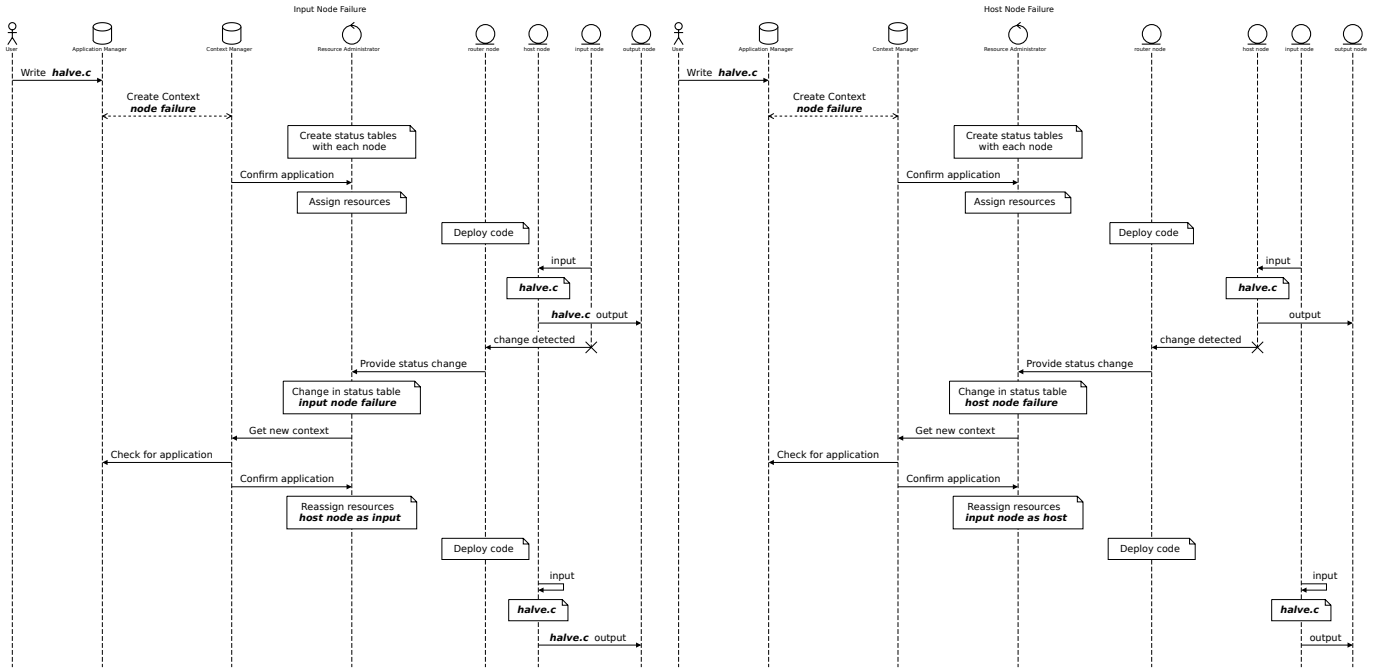

Fig. 8: Sequence Diagram for CAP

standard) has been used. This is intended to provide scalability and interoperability. There is no clear performance results on a number of supporting nodes or network data throughput. PyoT does not support group abstractions of 6LoWPAN and CoAP which can decrease overall performance of WSN. PyoT is also vulnerable for security issues due to CoAP implementation, that does not support Datagram Transport Layer Security (DTLS).

Python-based framework for ubiquitous networked sensors (PyFUNS) [16], is built based on PyMite facilitates reprogramming of virtual machines. That provides ease in application level programming by abstracting low level and networking functionalities. PyFUNS modifies network parameters according to its application counterpart and calibrates the network energy efficiency. PyFUNS is restricted to ContikiOS application level reconfigurations. The confined RESTful architecture i.e. the CoAP protocol does not offer transportation layer security and interoperability. The support of IPv6 and CoAP protocol nearly saturates the RAM of WiSMote, that restricts PyFUNS usage in complex tasks. Moreover, the trade-offs between energy consumption with script execution time, and saturation of RAM with communication failures were not studied.

Another contribution with similar focus is Code in the Air [6]. CITA enables programmer to write re-usable code which can be deployed in different energy based contexts. CITA builds a catalog for the applications and client-server approach is used to deploy the code.

These are some of the recent efforts, which can provide a limited support for context-awareness. Many of these also rely on CoAP operations, similar to the work presented in this paper. While CoAP is not most secure solution, it is supported by many other tools available for mobile systems, which makes it very popular.

VII. Conclusion

This paper proposes Context-Awareness Programming (CAP) for Cyber Physical Systems and describes its essential features. CAP combines Python scripts with a Django based web application to provide autonomous adaptations for different contexts. The proposed approach, CAP, is demonstrated using TelosB sensor nodes and TinyOS software. CAP includes three essential features, Abstraction, Modularity and Mobility. In author's best knowledge this has not been done by any of the previous works.

The current implementation of CAP is limited to proof of concept. In the future,It is important to provide faster adaptation with switching between multiple contexts. Another future goal would be to include complex data from multiple devices such as a calendar or activity of a user and recognize context with such data. Basic machine learning algorithms can help in recognizing such context with more reliability. Efforts to evaluate such contribution against existing context-aware work in embedded systems would be critical as well.

In addition, this work can serve as a building block for a complete framework to support context-awareness in WSNs. Such a framework would include management of applications, resources, and network. Each part proposes its own research problems. For example, to design resource management, it may be required to create a dynamic schedule for context-awareness to provide assurance on deadlines for different applications.

ACKNOWLEDGEMENT

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (CEC/04234); also by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within project(s) POCI-01-0145-FEDER-029074 (ARNET); and also by the EU ECSEL JU under the H2020 Framework Programme, within JU grant nr. 737422 (SCOTT project, www.scottproject.eu).

REFERENCES

[1] P. Leitão, A. W. Colombo, and S. Karnouskos, "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges," *Computers in Industry*, vol. 81, pp. 11–25, 2016.

[2] Z. Specification, "Zigbee alliance," *ZigBee Document 053474r06, Version*, vol. 1, 2006.

[3] Z. Shelby and C. Bormann, *6LoWPAN: The wireless embedded Internet*. John Wiley & Sons, 2011, vol. 43.

[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, IEEE, 2004, pp. 455–462.

[5] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*, Springer, 2005, pp. 115–148.

[6] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Code in the air: Simplifying sensing and coordination tasks on smartphones," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems and Applications (HotMobile)*, San Diego, California: ACM, 2012.

[7] D. Alessandrelli, M. Petraccay, and P. Pagano, "T-res: Enabling reconfigurable in-network processing in iot-based wsns," in *IEEE International Conference on Distributed Computing in Sensor Systems, 2013*.

[8] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *First Workshop on Mobile Computing Systems and Applications, 1994.*.

[9] G. Settanni, F. Skopik, A. Karaj, M. Wurzenberger, and R. Fiedler, "Protecting cyber physical production systems using anomaly detection to enable self-adaptation," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, IEEE, 2018, pp. 173–180.

[10] A. Ward, A. Jones, and A. Hopper, "A new location technique for the active office," *IEEE Personal communications*, vol. 4, no. 5, pp. 42–47, 1997.

[11] H. W. Gellersen, A. Schmidt, and M. Beigl, "Multi-sensor context-awareness in mobile devices and smart artifacts," *Mobile Networks and Applications*, vol. 7, no. 5, pp. 341–351, 2002.

[12] P. J. Brown, J. D. Bovey, and X. Chen, "Context-aware applications: From the laboratory to the marketplace," *IEEE personal communications*, vol. 4, no. 5, pp. 58–64, 1997.

[13] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *International symposium on handheld and ubiquitous computing*, Springer, 1999, pp. 304–307.

[14] G. Fortino, R. Giannantonio, R. Gravina, P. Kuryloski, and R. Jafari, "Enabling effective programming and flexible management of efficient body sensor network applications," *Human-Machine Systems, IEEE Transactions on*, 2013.

[15] G. P. P. Luca Mottola, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, 2011.

[16] S. Bocchino, S. Fedor, and M. Petracca, "Pyfuns: A python framework for ubiquitous networked sensors," in *European Conference on Wireless Sensor Networks*, Springer, 2015, pp. 1–18.

[17] M. Wasilak. (2015). Txthings, [Online]. Available: https://github.com/siskin/txThings/.

[18] M. Kovatsch, "Demo abstract: Human–coap interaction with copper," in *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems*, 2011.

[19] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks*, ACM, 2007.

[20] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: A methodology for architecture-independent programming of networked sensor systems," in *Proceedings of the 2005 Workshop on End-to-end, Sense-and-respond Systems, Applications and Services*.

[21] A. Elsts, F. H. Bijarbooneh, M. Jacobsson, and K. Sagonas, "Profun tg: A tool for programming and managing performance-aware sensor network applications," in *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*, IEEE, 2015, pp. 751–759.

[22] A. Azzara, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano, "Pyot, a macroprogramming framework for the internet of things," in *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, 2014, pp. 96–103. DOI: 10 . 1109 / SIES . 2014 . 6871193.

[23] V. Gupta, J. Kim, A. Pandya, K. Lakshmanan, R. Rajkumar, and E. Tovar, "Nano-cf: A coordination framework for macro-programming in wireless sensor networks," in *8th Annual IEEE SECON*, 2011, pp. 467–475.