FEDERAL UNIVERSITY OF CEARÁ

CENTER OF SCIENCE

COMPUTER SCIENCE DEPARTMENT

MASTER AND DOCTORAL PROGRAM IN COMPUTER SCIENCE

MARCIO ESPINDOLA FREIRE MAIA

SYSTEM SUPPORT FOR SELF-ADAPTIVE CYBER-PHYSICAL SYSTEMS

FORTALEZA

2015

MARCIO ESPINDOLA FREIRE MAIA

SYSTEM SUPPORT FOR SELF-ADAPTIVE CYBER-PHYSICAL SYSTEMS

Doctoral thesis presented to the Master and Doctoral Program in Computer Science from the Center of Science in the Federal University of Ceará, as part of the requirements to obtain the degree. Concentration Area: Computer Science

Supervisor: Rossana Maria de Castro Andrade

FORTALEZA

2015

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M187s Maia, Marcio Espíndola Freire.
System Support for Self-Adaptive Cyber-Physical Systems / Marcio Espíndola Freire Maia. – 2015.
130 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2015.
Orientação: Profa. Dra. Rossana Maria de Castro Andrade.

1. Internet of Things. 2. Middleware. 3. Framework. 4. Self-Adaptation. I. Título.

CDD 005

MARCIO ESPINDOLA FREIRE MAIA

SYSTEM SUPPORT FOR SELF-ADAPTIVE CYBER-PHYSICAL SYSTEMS

Doctoral thesis presented to the Master and Doctoral Program in Computer Science from the Center of Science in the Federal University of Ceará, as part of the requirements to obtain the degree. Concentration Area: Computer Science

Aproved in: October 3rd, 2015

EXAMINATION BOARD

Rossana Maria de Castro Andrade (Supervisor)
Federal University of Ceará

Markus Endler
Pontifical Catholic University of Rio de Janeiro
(PUC-RJ)

Elias P. Duarte Jr.
Federal University of Paraná (UFPR)

Danielo Goncalves Gomes
Federal University of Ceará (UFC)

Windson Viana de Carvalho
Federal University of Ceará (UFC)

I dedicate this thesis to my wife Marina, for all the love and support. Also to our beautiful daughter Nina, who fulfills our days with joy. I love you both very much.
Thank you for everything.

ACKNOWLEDGEMENTS

I would like to say Thank You to Professor Rossana M. C. Andrade, for all the advising and counseling during the execution of this thesis.

To all the Professors from the Group of Computer Networks, Software Engineering and Systems (GREat), my appreciation, for being always available when I needed. I would like to extend my gratitude to everyone at GREat.

Thank you very much to all my friends at Campus de Quixada, from Federal University of Ceará.

My thanks to my parents, Winston and Olga. Everything that I am today I owe it to you. To my brothers, Renato and Diane, for all the lifetime company.

RESUMO

Dispositivos com capacidade de processamento e comunicação estão cada vez mais embutidos no ambiente físico, gerando informações sobre os usuários e suas interações com o ambiente. Essa proximidade entre usuários, objetos do mundo físico e serviços está permitindo o aparecimento dos sistemas cyber-físicos (Cyber-Physical Systems - CPS). Neles, a quantidade de dispositivos cresce muito rapidamente e mecanismos que permitam a gerência das interações entre eles são necessários. Assim, essa tese de doutorado propõe um middleware, chamado CyberSupport, que fornece um suporte para o desenvolvimento e execução baseado em camadas de software auto-adaptáveis para CPS. O CyberSupport possui na sua camada mais inferior mecanismos para permitir a comunicação e coordenação entre os diversos dispositivos que formam CPS. A principal contribuição dessa camada está relacionada com as interações entre dispositivos de forma desacoplada e auto-adaptativa. Na camada superior, primitivas de execução fornecem acesso aos recursos do ambiente através de interfaces bem definidas, com especial atenção para facilitar a adaptação do funcionamento da aplicação e do CyberSupport. Essas interfaces são utilizadas para a criação de sistemas auto-adaptativos para CPS. O CyberSupport foi avaliado utilizando métricas de desempenho em dispositivos reais, e também métricas de qualidade de código, comparando aplicações desenvolvidas com e sem o CyberSupport. As métricas de desempenho avaliaram 5 algoritmos de roteamento e 5 tecnologias de comunicação desenvolvidos utilizando o CyberSupport, de acordo com o tempo médio de entrega de uma mensagem, a quantidade total de mensagens entregues e a taxa de perda de mensagens. Já as métricas de qualidade de código compararam 3 aplicações desenvolvidas com e sem o CyberSupport, de acordo com métricas de acoplamento de código. Os resultados mostraram uma redução de 3% no acoplamento de código quando o CyberSupport é utilizado.

Palavras-chave: Internet das Coisas. Sistemas Cyber-Físicos. Middleware. Auto-adaptação.

ABSTRACT

Devices with computational and communication capabilities are pervaded throughout the physical environment, generating information about users and their interactions to enrich the virtual world. This proximity between physical and cyber worlds has allowed the appearance of Cyber-Physical Systems (CPS), formed by users and devices (physical) interacting with services, components and applications (cyber). When the number of physical and cyber entities increase, so does the need for mechanisms to specify and manage their interactions. In that direction, this doctoral thesis proposes CyberSupport, an adaptable software stack to offer system support for self-adaptive CPS. The thesis contribution is based on layers: on the bottom, the communication and coordination layer is responsible for a modular, uncoupled and adaptable infrastructure to permit access and control of environment resources, along with message exchange and interaction of decentralized devices. On the top layer, the execution and adaptation layer permits to specify and implement monitoring and execution functionalities to support the creation of self-adaptive CPS. CyberSupport is evaluated according to performance metrics using real devices, along with design metrics using existing applications before and after using CyberSupport. The performance metrics evaluate 5 routing algorithms and 5 communication technologies implemented using CyberSupport according to message delivery and message loss metrics. Additionally, the design evaluation reimplementes 3 existing applications and compares the implementations with and without CyberSupport according to coupling metrics. The results shows an improvement of around 30% considering the design metrics.

Keywords: Internet of Things. Cyber-Physical Systems. Middleware. Self-Adaptation.

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

CPS	Cyber-Physical Systems
SPL	Software Product Line
$MAPE-K$	Monitor, Analyze, Plan, Execute, Knowledge
QoS	Quality of Service
CIB	Context Information Base
CAC	Context-Actuator Component
SCA	Service-Component Architecture
AC	Autonomous Component
GCM	Google Cloud Messaging
$SDDL$	Scalable Data Distribution Layer
CDC	Concern Diffusion over Components
CDO	Concern Diffusion over Operations
$CDLoC$	Concern Diffusion over Lines of Code
CBC	Coupling Between Components
NFC	Near-Field Communication
BLE	Bluetooth Low Energy
$UMTS$	Universal Mobile Telecommunications System
LTE	Long-Term Evolution
$Wi-Fi$	Wireless Fidelity
NAR	Number of Address References
NOI	Number of Online Interactions
NSI	Number of Synchronous Interactions
API	Application Programming Interface
CBD	Component-Based Development
ADL	Architecture-Description Language
$OSGi$	Open Services Gateway Initiative
RTT	Round-Trip Time

CONTENTS

1 INTRODUCTION

This thesis proposes a framework called CyberSupport: System Support for Cyber-Physical Systems. In Section 1.1, the problem statement is discussed. Section 1.2 presents a running example that motivated the creation of CyberSupport. Section 1.3 details the hypothesis and research questions. Section 1.4 presents the goals of this thesis. Section 1.5 describes the methodology employed during the creation of CyberSupport and Section 1.6 lists the remaining structure of this document.

1.1 Contextualization and Problem Statement

The software engineering body of knowledge is evolving to tame the increasing complexity of modern software. When Mark Weiser coined the Ubiquitous Computing definition in his seminal paper entitled *The computer of the 21st century* (WEISER, 1999), he was predicting computation to be embedded into the environment, so seamlessly as to become indistinguishable from it. Since its conception, the Ubiquitous Computing definition has evolved to a broader concept called Cyber-Physical Systems (CPS), in order to reach any Cyber and Physical interacting device (CONTI *et al.*, 2012).

CPS can be defined as *"an orchestration of computers and physical systems. Embedded computers monitor and control physical processes, with feedback loops where physical processes affect computations and vice versa"* (LEE, 2015). Now, CPS are no longer composed of a small number of interacting computers. Instead, they are formed by a large number of computing devices, with different hardware and software characteristics. These devices vary from simple sensors and actuators embedded into the environment, such as physical objects, wearable items and home appliances, to fully-equipped computers like personal computers, smartphones and tablets (ANTSAKLIS, 2014). Figure 1 shows examples of cyber-physical systems.

Aggregating all available devices, CPS are present in several application domains, such as ambient intelligence (HELAL *et al.*, 2005; BRUMITT *et al.*, 2000), mobile visit guide (GRUN *et al.*, 2008; MOBILINE..., 2013), e-Health (RODRÍGUEZ *et al.*, 2009; MAITLAND *et al.*, 2011), mobile social networks (SCHUSTER *et al.*, 2013; MAIA *et al.*, 2012) and smart transportation systems (DIMITRAKOPOULOS; DEMESTICHAS, 2010). Although each domain presents specific development and execution requirements, they somehow share the

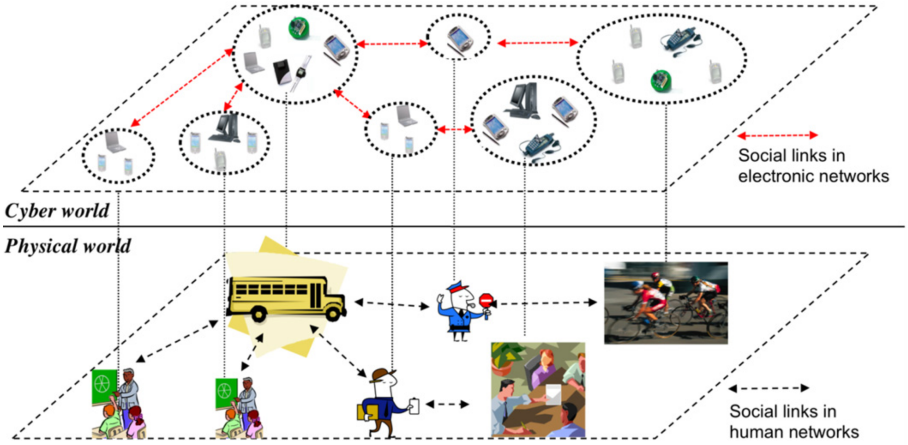

Figure 1 – Cyber-Physical Systems. Extracted from (CONTI *et al.*, 2012).

following challenges (PERERA *et al.*, 2014a):

1. **Decentralization**, whereas the number of interacting devices and administrative domains varies unpredictably, a single centralized control should be avoided.
2. **Device and user mobility**, making it difficult to define at development-time available resources at runtime.
3. **Uncoupled and spontaneous interaction**, caused by the decentralization and unpredictable mobility of devices and users, all interactions should minimize coupling.
4. **Interoperability**, since the interacting device characteristics varies considerably, interaction should be based on common formalisms.
5. **Context-awareness and adaptability**, in order to sense the execution environment and adapt to the changing circumstances.

To implement CPS considering these challenges, developers make use of supporting technologies, which are wireless communication technologies, sensor and actuator platforms, context management infrastructures, programming paradigm (e.g. components, services) and coordination mechanisms (e.g. events, tuples, direct message exchange) (MAIA *et al.*, 2009).

Available supporting technologies (communication, sensors/controller access, context management, programming and interaction) define the solution space. For instance, wireless communication can be implemented using Bluetooth, Wi-Fi or Near-Field Communication (NFC); the programming dimension may use services, components or tasks, among others; while the coordination dimension could use events or tuples. The definition of the solution space is part of the CPS development.

CPS development has the following specific characteristics that must be considered during the entire development cycle (BARESI; GHEZZI, 2010; AUTILI *et al.*, 2012): i) com-

plexity to specify at development time all possible application scenarios, in order to define the most efficient technologies from the solution space; and ii) incapacity to perform maintenance actions at runtime to change the solution space, due to the small time-frame required by CPS and their inherent decentralization.

If these two characteristics are not incorporated into the development phases, this could generate rigid design-time architectural decisions, incapable of dealing with evolving requirements and unpredicted situations, very frequently found in running CPS. Thus, CPS development practices must incorporate techniques to change software structure and behavior more easily.

Among these techniques, self-adaptation has been used to create systems capable of adapting behavior and structure to deal with changes in the execution environment, with minimum user and developer intervention (KRAMER; MAGEE, 2007). Hence, self-adaptation is a development technique to manage the running system, presenting some degree of autonomy. It is based on system models, describing functional and non-functional goals. At runtime, these goals are translated to execution modules that implement each requirement.

Self-adaptation is achieved using a feedback control loop (KRAMER; MAGEE, 2007; KRUPITZER *et al.*, 2014), similar to robotic systems, performing four activities: i) perceiving the environment, ii) analyzing if the system is in a correct or exceptional state, iii) planning adaptive actions to correct exceptional states; and iv) executing the planned actions. Feedback control loops are usually implemented using centralized approaches, what leads to inefficient solutions when the size and complexity of the system increases.

The implementation of self-adaptation functionalities usually uses a middleware layer, abstracting away from developers the complexity to implement primitives such as sensors access and control, communication, resources discovery, among other functionalities (MAIA *et al.*, 2009; BLAIR *et al.*, 2011; ZHOU *et al.*, 2014).

CPS in execution are composed of middleware and application (WEYNS *et al.*, 2013). Then, the same issues concerning design-time decisions are applied to the middleware layer, where not only the application itself must adapt to changes in the execution environment, but also the middleware layer should be able to change its behavior as well. Hence, self-adaptation can be used both in the application and middleware layer.

Applying self-adaptation to running CPS blurs the boundary between application and middleware layer (AUTILI *et al.*, 2012). The challenge is to create CPS using a middleware

support flexible enough to implement applications capable of adapting their behavior, reusing existing middleware primitives (e.g. sensor/actuator access, communication), and permitting the adaptation of the middleware primitives as well.

That flexibility is achieved using two separate approaches: i) fostering localized interactions, where any centralization should be avoided (WEYNS *et al.*, 2010); and ii) minimizing coupling among all interacting entities (FAISON, 2011).

1.2 Running Example

Imagine a situation where the user is carrying his/her smartphone and wearing a smartwatch. The user is immersed in a physical environment filled with computing devices with processing and communication capabilities. These two classes of devices (user and environment) interact to offer a self-adaptive mobile user guide. The guide is responsible for conducting the user throughout the environment, showing a map of where the user is, along with information about other users, the environment itself and cyber/physical objects belonging to that environment (e.g. art pieces, multimedia files). Mobile visit guide may have other functionalities such as to coordinate the movement of several users by exchanging messages directly between the users in a group-based movement. Some visit guides may increase in size, and the number of users and cyber/physical objects push past several thousand items (MAMEI; ZAMBONELLI, 2005).

An example of a self-adaptive mobile user guide is derived from a Software Product Line (SPL) called Mobiline[1], created in our research group during the UbiStructure project[2]. SPL is a reuse-based software development approach to create software based on reusable artifacts, by modeling an application domain, and then describing mandatory and optional characteristics of every application derived from that domain. The Mobiline SPL specifies the domain of context-aware mobile visit guides, where the self-adaptive mobile user guide called GREat Tour is one product of Mobiline. More information about the Mobiline SPL can be found at the Mobiline project web site or in the main publication of the project (MOBILINE..., 2013).

The GREat Tour SPL implementation was the starting point. It was reimplemented to use the contributions made during this doctoral thesis. Thus, the main functionality provided by GREat Tour is the ability to locate the user within the building where the GREat laboratory is located. Two indoor location approaches were implemented. The first based on QR-Code (one

[1] http://mobiline.great.ufc.br/
[2] http://ubistructure.great.ufc.br/index.php/en/

qr-code per room) and the second based on NFC Tags (each tag contained the room id). Based on the user location in the building, a map shows where the user is located. Once the user enters a room, he/she receives information about that room, such as other users present in that room and available cyber/physical objects. Information about users and objects can be presented based on text, photos and video. The interaction between users, and also between users and physical objects is an important requirement that must be offered to the applications.

The actors that interact with the application are User, Item and Environment. Each Environment has a Service Discovery mechanism to inform available services at a given moment (e.g. information about an Environment or Item). These services were described using a Service Discovery mechanism.

To start the application, the User authenticates with a username and password. Once authenticated, a user may visualize a map of the entire guide; choose to view information according to his profile (which was previously defined). Once an Environment is chosen, the user may select to view a list of items and choose access to information about a given item.

Context management is responsible for acquiring and providing information about the user indoor location. As a specific environment is entered, the user may obtain information about its location, items and the presence of other users. The behavior of the application is also affected by device restrictions, such as remaining battery or available libraries.

Another functionality that must be offered to application developers is the ability to exchange messages between devices and to coordinate the execution of a given functionality (e.g. download an item or obtain the list of existing users).

Existing users, available items and execution context may vary unpredictably, making it difficult to rely entirely on design-time decisions. Opposingly, the application must discover at runtime the available users and items, along with device capabilities. Only then, adapt its behavior accordingly.

1.3 Hypothesis and Research Questions

Considering the discussion on CPS presented in sections 1.1 and 1.2, this doctoral thesis investigates the following hypothesis:

The development of self-adaptive cyber-physical systems requires an uncoupled and decentralized execution infrastructure.

From this hypothesis, three research questions are derived, as follows:

Research Question 1: What are the main requirements to develop self-adaptive cyber-physical systems?

Research Question 2: What are the main challenges to implement self-adaptive cyber-physical systems?

Research Question 3: How coupling affects the development and execution of decentralized self-adaptive cyber-physical systems?

1.4 Goals and Activities

This doctoral thesis proposes a support infrastructure for the development and execution of self-adaptive cyber-physical systems called CyberSupport: System Support for Cyber-Physical Systems. Its main goal for self-adaptive CPS is to improve uncoupling and to provide decentralization. In order to fulfill this goal, the following milestones were set:

Activity 1: To develop a modular technology-independent communication infrastructure for building decentralized CPS.

Activity 2: To develop a decentralized and uncoupled coordination infrastructure.

Activity 3: To develop a component management infrastructure for implementing access to environment sensors and actuators.

Activity 4: To develop a supporting infrastructure for the development and execution of self-adaptive cyber-physical systems.

Activity 5: To evaluate how uncoupled the developed infrastructure is.

1.5 Research Methodology

The methodology for the development of this doctoral thesis is described as follows:

1. To study the concepts related to the development of self-adaptive cyber-physical systems.
2. To perform a bibliographic review to identify the requirements for self-adaptive cyber-physical systems and the existing middleware infrastructure.
3. To investigate how uncoupling can affect the development and execution of self-adaptive CPS.
4. To identify relevant case studies, in order to illustrate the complexity to develop self-

adaptive cyber-physical systems and to validate the identified requirements.

5. To compare the existing work listed during phases ii, iii and iv.
6. To create an execution infrastructure considering phase v.
7. To evaluate the created infrastructure.

1.6 Document Organization

This chapter presented the issues motivating the creation of an execution support infrastructure for the development of self-adaptive cyber-physical systems called CyberSupport: System Support for Cyber-Physical Systems. It also presented the hypothesis and research questions, along with its goals and activities as well as the thesis methodology. The remainder of the document is organized as follows:

Chapter 2 presents the main topics related to this doctoral thesis, discussing the main challenges and requirements to build self-adaptable systems, considering specially decentralization. Another topic detailed in this chapter is the requirements to implement CPS.

Chapter 3 presents the related work separated in layers, and then a work proposing an integrated solution, like the one proposed in this thesis.

Chapter 4 presents the main contributions of this doctoral thesis, divided in two layers. The bottom layer, called communication and coordination layer, is responsible for a modular, uncoupled and adaptable infrastructure to permit access and control of environment resources, along with message exchange and interaction of decentralized devices. On the top layer, the execution and adaptation layer permits to specify and implement self-adaptation based on components and reactions to environment events.

Chapter 5 presents the performance evaluation executed for each layer. Additionally, the evaluation using couplings metrics is performed, to verify the coupling level among system modules and interacting entities.

Chapter 6 presents the concluding remarks, analyzing the research questions and the design decisions to build the contribution as it is now. It also lists the publications as a result of this doctoral thesis.

2 DECENTRALIZED SELF-ADAPTIVE SYSTEMS

This chapter presents the concepts related with the construction of self-adaptive CPS, beginning with the specification of architecture-based self-adaptation and how decentralization impacts on the creation and management of CPS. Then, it discusses the issues for implementing self-adaptive CPS from the perspective of existing technologies. The goal is to understand the requirements for self-adaptation in CPS and to describe the foundation for the creation of a support infrastructure for self-adaptive CPS.

2.1 Introduction

Autonomic computing has been proposed to tame the increasingly complexity to manage and maintain large IT systems (KEPHART; CHESS, 2003). Suggested in 2001 by IBM's Research Vice President Paul Horn to be used in the development and management of software systems, it is becoming a viable alternative to handle the inherent dynamism of CPS.

In such systems, composed of several distributed devices, autonomic behavior is implemented specifying high-level system goals using system models. These models map system goals to actions executed independently by each device. Although a promising concept, to develop fully autonomic CPS is is still a challenge (KRUPITZER *et al.*, 2014).

Rather, self-adaptive CPS incorporate concepts proposed for autonomic systems, but it still keeps developers and users in charge of managing high-level system execution (MEDVIDOVIC; EDWARDS, 2010). Using self-adaptation, system developers and managers define high-level execution goals, requirements, or Quality of Service (QoS) restrictions and the system performs all necessary adaptations at run-time (LEMOS *et al.*, 2013).

Each device implementing self-adaptive behavior is called self-adaptive entity, which is responsible for managing resources under its control. Management actions are executed on managed resources to configure, optimize, heal and protect the running system, described as follows (HUEBSCHER; MCCANN, 2008):

Self-Configuration permits the system to configure itself based on high-level goals, specifying system requirements. On the running system, it changes execution modules and installs different implementations, in order to meet changing requirements.

Self-Optimization permits the system to proactively optimize the use of its managed resources, changing execution parameters and trying to improve the overall performance or QoS.

Self-Healing allows the system to detect, analyze and fix execution errors and failures. In that direction, fault-tolerance is an important technique to implement self-healing, reacting to erroneous system states.

Self-Protection is a property to let the system to proactively protect itself from attacks or incorrect changes. This property is important to implement security or privacy requirements, by anticipating possible security issues from taking place.

Each self-* property is implemented based on self-management concepts, responsible for coordinating, maintaining and evolving the running system. Thus, a control loop, similar to what is used in robotic systems, is used to implement self-adaptation (IBM Corp., 2004).

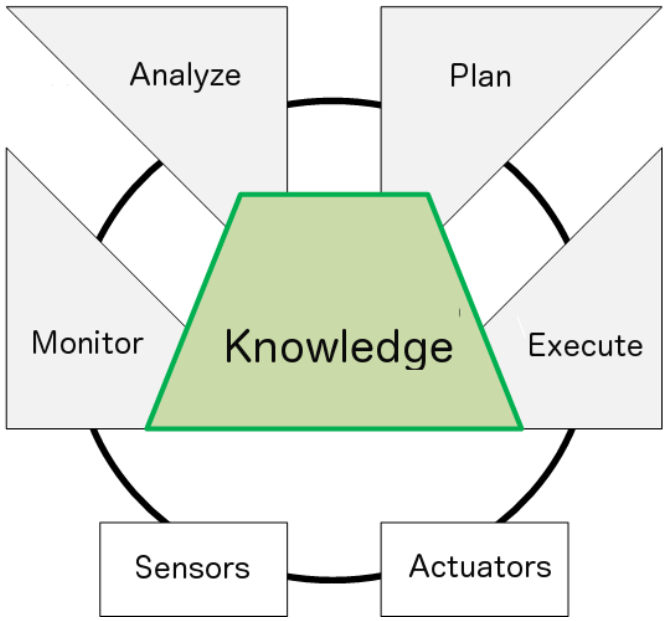

Figure 2 – MAPE-K control loop. Adapted from (IBM Corp., 2004)

Figure 2 presents the MAPE-K control loop presented in (IBM Corp., 2004), formed by four stages: Monitor, Analyze, Plan and Execute (MAPE), along with the system knowledge (K), representing the system models describing its goals, requirements and restrictions.

The Monitor phase is responsible for accessing sensors present in the execution environment to collect, filter and aggregate information describing available resources. Then, information representing the running system is described using an execution model and passed on to the Analyze function.

Upon receiving the execution model, describing its current state, another model describing the expected state is used. The current and desired states are compared to define

whether the system requires any adaptation or not. Adaptation decision uses also models describing how the system and execution environment evolves. The outcome is an adaptation request to the Plan function.

The Plan phase defines an execution plan responsible for putting the system back onto a desired state. The execution plan is a sequence of actions performed on the system and its resources, generated using a system model, along with the current state. If no path (sequence of actions) from the current to the desired state can be found, the system must receive external information from the user, developer or external system on how to react. Once an execution plan is defined, the system executes the defined actions.

Actions defined by the execution plan change system and environment state using actuators. Once the plan is carried out, the control loop is executed again, to define if the system did entered the expected state. If learning approaches are available, the system models present in the knowledge base can be updated.

Knowledge base is a dictionary containing the entire system models, comprised of policies, requirements, states, goals and actions. Such information is described using knowledge representation mechanisms (MORIN *et al.*, 2009; ALFÉREZ; PELECHANO, 2012; BENCOMO, 2009).

2.2 Reference Architecture

CPS are usually composed of several independent uncoupled subsystems interacting by message passing (WEYNS *et al.*, 2013). As mentioned before, self-adaptation is useful to facilitate their management and to guarantee they actually deliver the promised functionalities.

Based on the MAPE-K paradigm, self-adaptation is accomplished monitoring a set of system resources using sensors, analyzing and planning adaptation using system models, and executing adaptation actions upon the environment using actuators. This control loop is based on the following premises: i) updated and consisted view of the managed resources; ii) existence of system models describing their execution and desired properties; and iii) possibility to act upon environment resources.

The challenge in CPS lies on guaranteeing these premises, mainly for three reasons. First, as the number of devices increases, it also increases the adversities to guarantee a consistent and updated view of the entire system. Second, to process that view and to infer relevant information is time-consuming and impractical to be used in models with a large number of

system states. Third, to permit the system to scale and handle the inherent dynamism of CPS, decentralization is mandatory. However, decentralization imposes restrictions on inference using information about the entire system.

When decentralization increases, solutions based on a single, or a few, control entities tend to become inefficient. Agent-based (WOLF; HOLVOET, 2003; WOLF; HOLVOET, 2005) or nature-inspired approaches (ZAMBONELLI; VIROLI, 2011) have been proposed to implement self-adaptation in fully distributed scenarios. Now, each entity has its own behavior and global goals are achieved by the emergence of collective actions.

Although it is relatively simple to specify system-wide goals, to map them to individual behavior is a challenge. Another complex issue is to provide assurances on a global basis. Finally, even though agent-based and nature-inspired solutions have successfully been used before, they are problem-specific and cannot be generalized to other application domains (KRUPITZER *et al.*, 2014).

Self-adaptation can be specified based on system modules and interactions, defining its overall architecture. The main benefits of designing self-adaptive systems based on their architecture are three-fold: i) generality, since system architecture can be applied to several systems from the same application domain; ii) abstraction and modularization, since each module hides implementation details and only high-level interactions are described; and iii) reuse, since existing implementation of architectural elements can be reused.

Architecture-based self-adaptation borrows its concepts from robotic systems using the sense-plan-act (DOBSON *et al.*, 2006) approach. In general, their architecture is divided in layers (GARLAN; SCHMERL, 2002; IBM Corp., 2004): 1) Monitor and control, responsible for accessing information about and acting upon the execution environment; 2) Management, responsible for analyzing the current and desired system state, along with executing corrective/adaptive actions; 3) Goal, responsible for generating execution plans to guide the system towards its goals. Figure 3 shows an architecture describing these three layers.

The monitor and control layer (called Component Control in Figure 3) is responsible for collecting execution information about system modules, notifying upper layers of events happening in the environment and acting upon these modules. These modules are often encapsulated as components. Hence, access to sensors and actuators takes place using interfaces provided by components. Finally, interaction between resources happens using connectors.

In this layer, algorithms to collect and disseminate information should consider

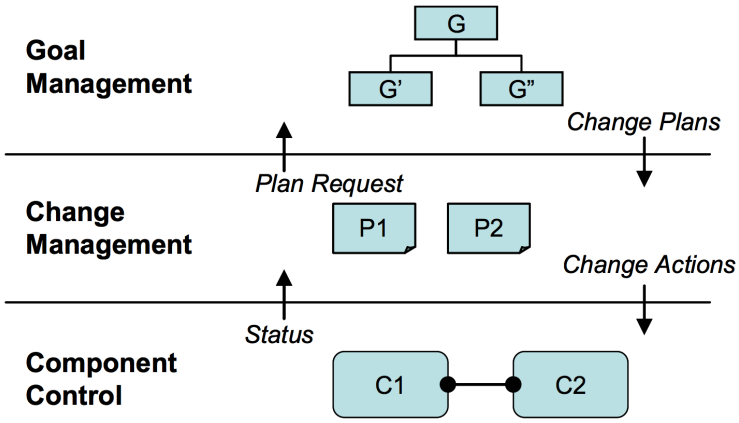

Figure 3 – Architecture-based self-adaptation. Extracted from (KRAMER; MAGEE, 2007)

scalability as design principle. Additionally, to handle the inherent dynamism of CPS, interaction between modules should minimize coupling between modules.

The Management layer (called Change Management in Figure 3) is responsible for executing the plan defined by the Goal layer. Execution is carried out accessing sensors and actuators present in the monitor and control layer. Additionally, since each plan is tailored to be executed in a specific context, when changes in the lower layer are detected, making the current execution plan inadequate, the Management layer requests a new execution plan from the Goal layer.

In decentralized systems, information about the execution environment comes from several distributed sources. Considering the dynamism and heterogeneity of CPS, mechanisms to describe and discover available resources at runtime should be used (MAIA *et al.*, 2009). Additionally, at this layer, interoperability should also be a concern (BLAIR *et al.*, 2011; AUTILI *et al.*, 2012).

The Goal layer (called Goal Management in Figure 3) considers the current and final state of the system and generates an execution plan to make the system enter in a desired state. This layer is requested in two scenarios: i) the system entered a state where the current execution plan cannot handle, and the Management layer requested a new execution plan; or ii) the overall system goals or requirements changed and the knowledge base was updated, requiring a new execution plan to move the system from the current state to the new goals.

Decentralization imposes extra complexity to the specification of the Goal layer, since high-level goals are not easily mapped to an execution plan executed on several distributed resources at the lower layers. Additionally, when the decentralization increases, individual

resources may have conflicting goals (KRAMER; MAGEE, 2007).

The MAPE-K reference architecture (shown in Figure 2) and the layer-based approach (shown in Figure 3) are useful to specify and implement self-adaptation, presenting modularity and reusability as important contributions. Thus, self-adaptation is carried out sensing and acting on the execution environment, based on high-level goals. That approach divides the system in two parts (WEYNS *et al.*, 2013): i) Managed system, formed by the subsystems implementing the functional requirements; and ii) Management system, inspecting system execution and adapting/modifying the Managed system according to the MAPE-K or layer-based architecture.

In Figure 3, the managed system is present in the lower layer (Component Control), whereas the management system is formed by the two top layers (Change and Goal). Management actions are executed accessing information about system execution, modifying parameters on the lower components or changing the components themselves.

2.3 Decentralized Control Patterns

In CPS, usually, a system is composed of several interacting subsystems. In that scenario, self-adaptation takes place internally, in each subsystem, and externally, throughout subsystems. Hence, to specify the system architecture of several interacting self-adaptative subsystems, the MAPE-K control loop must accommodate the coordination among subsystems.

Depending on the system architecture, coordination takes place in different levels within the MAPE-K loop. To understand the existing solutions to for implementing coordination among several self-adaptive subsystems, Weyns et al. (WEYNS *et al.*, 2013) present a compilation of reference architectural patterns for decentralized self-adaptive systems.

2.3.1 Coordinated Control Pattern

In highly distributed scenarios, solutions to control self-adaptation must avoid any type of centralization, which may induce high costs to collect and process the information; or the system spans across administrative domains without trust assurances, in which adaptation requests among domains are not trusted.

The coordinated control pattern has one MAPE-K loop for each subsystem. Each phase (Monitor, Analyze, Plan, Execute) coordinates its execution with other instances of the

same phase using message passing. Figure 4 shows the coordination among MAPE-K loop instances. For instance, M components exchange execution information and A components decide on local adaptation requirements.

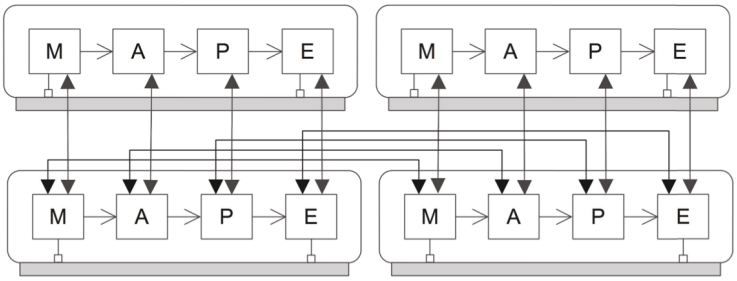

Figure 4 – Coordinated control pattern. Extracted from (WEYNS *et al.*, 2013).

The coordinated control pattern scales well, depending on the number of modules each MAPE-K phase needs to interact with. Additionally, it increases the robustness of the overall system, since no single point of failure exists. Finally, it reduces the computational overhead, since processing is distributed among subsystems. Although effective to deal with decentralization, scalability may be compromised if the number of interactions within phases is high. It also makes it more difficult to guarantee adaptation consistency and optimality.

2.3.2 Information Sharing Pattern

Another pattern of highly distributed scenarios is the information sharing pattern. It describes self-adaptation where each subsystem has a complete instance of the MAPE-K loop, adaptation occurs independently, but information to guide the adaptation of each subsystem is received from others subsystems. Here, the interaction between subsystems happens only to share execution information. Figure 5 shows the interaction among different MAPE-K loops based on the Monitoring phase. Analyze, Plan and Execute takes place independently.

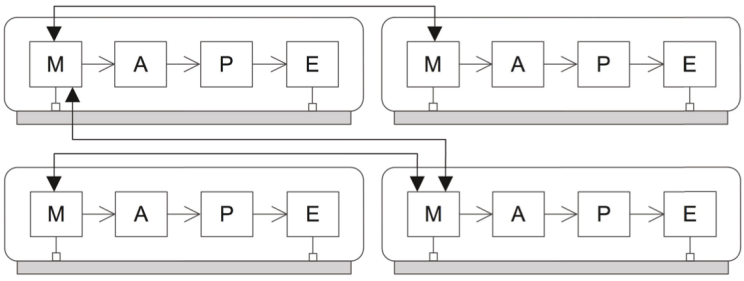

Figure 5 – Information sharing pattern. Extracted from (WEYNS *et al.*, 2013).

The information sharing pattern can be considered an instance of the coordinated control pattern, where only the monitoring phase interacts with other subsystems. Less interactions among distributed subsystems result in better support to scalability. Adaptation is achieved locally by the Analyze, Plan and Execute phases. The downside with local adaptation is conflicting or suboptimal adaptation decisions.

2.3.3 Master-Slave Pattern

In distributed situations, where system-wide assurances are necessary (WEYNS *et al.*, 2010), like performance or adaptation consistency, fully decentralized solutions are not indicated. To offer system-wide guarantees, the master-slave pattern can be used, centralizing the Analyze and Plan phases, but distributing amid subsystems the Monitor and Execute phases. Figure 6 shows the architecture of the master-slave pattern.

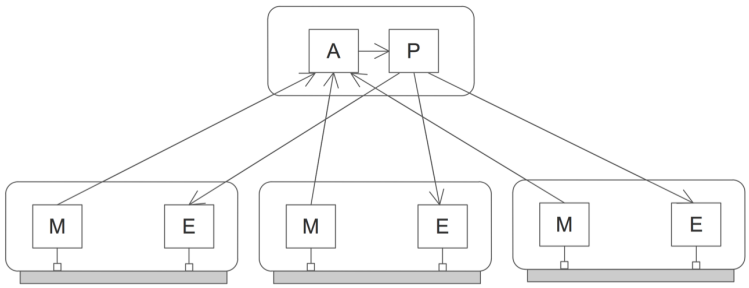

Figure 6 – Master-slave pattern. Extracted from (WEYNS *et al.*, 2013).

Using the master-slave pattern, distributed subsystems collect execution information and forward it to the centralized system, for decision making. When adaptation is required, the Plan phase is responsible for generating a new execution plan, which is executed individually by each distributed subsystems. By centralizing decision-making, system-wide assurances can be enforced. On the other hand, scalability is compromised, since the centralized decision-making and control may prohibit its use in large system. Also, robustness is a concern, as result of the single point of failure.

2.3.4 Regional Planning Pattern

A variation of the master-slave pattern is the regional planning pattern, in which each subsystem is responsible for the Monitor, Analyze and Execute phases, leaving out only the Plan phase to a regional leader. Here, monitoring and analysis are carried out independently in

each subsystem. For coordination among regions, the Plan components exchange messages and are responsible for generating an execution plan for each region, which is executed individually. Figure 7 shows the architecture of the regional planning pattern.

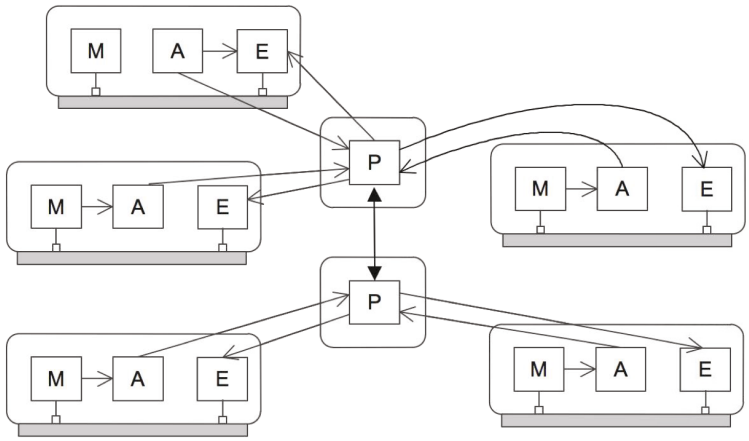

Figure 7 – Regional planning pattern. Extracted from (WEYNS *et al.*, 2013).

The main benefit of the regional planning pattern is to permit monitoring, control and execution among regions. This approach improves scalability when planning requests are kept locally, or when the number of planning requests that interacts with others regions is low. It also separates concerns, keeping each region responsible for its own adaptation. If the number of planning requests among regions increases, the regional planning pattern downgrades its efficiency.

2.3.5 Hierarchical Control Pattern

So far, separating the running system into management and managed systems, the MAPE-K loop acts upon the managed systems, adapting its behavior and structure. However, the management system itself is not adaptable. Is follows the adaptation rules specified at design time. Hence, to introduce adaptation on the management system, the Hierarchical control pattern uses at least two MAPE-K loops. The bottom loop is the normal MAPE-K loop previously presented. To change the bottom MAPE-K loop, another loop is responsible for monitoring the management system and adapting its behavior as well. It is as if the management system (bottom loop) becomes the managed system of the top loop. Figure 8 shows the architecture of the hierarchical control pattern. Each higher loop is responsible for managing the loop immediately bellow.

The hierarchical control pattern improves complexity management, considering it

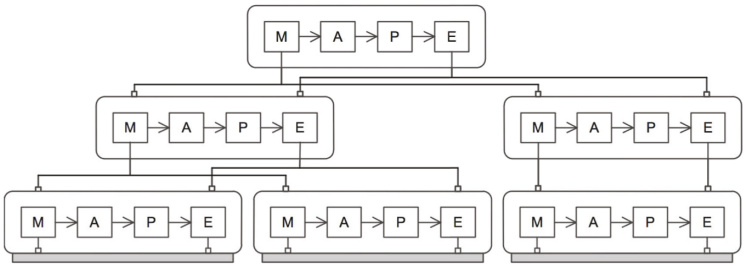

Figure 8 – Hierarchical control pattern. Extracted from (WEYNS *et al.*, 2013).

keeps the adaptive behavior organized in each level. This approach is similar to the one used by Kramer and Magee (KRAMER; MAGEE, 2007), presented in figure 3. The challenge here is to manage goals in each layer, specially when these goals affect each other (WEYNS *et al.*, 2013).

2.4 Developing Self-Adaptive CPS

Developing CPS presents specific challenges that need to be considered when devising a reusable solution, such as strong human presence, large number of personal and environment devices, different interacting systems with possible conflicting non-functional requirements (security, safety, interoperability, responsiveness), real-time, scalable and interoperable requirements (EVERS *et al.*, 2014).

To deal with these challenges, self-adaptation is a promising concept where the complexity to develop and manage is becoming impractical using current development techniques. Specifically in CPS, decentralization plays an important role and must be considered from scratch.

For each module participating in the decentralized MAPE-K loop, at least one of the four phases (Monitor, Analyze, Plan and Execute) must be present. For instance, in the coordinated control pattern (section 2.3.1, figure 2.3), each module is responsible for all four phases, where the interaction between modules occur within each phase. On the other hand, the master-slave pattern (section 2.3.3, figure 2.5), there are two different modules: 1) modules responsible for monitoring and executing, and 2) modules responsible for analyzing and planning. The interaction between modules takes places between the monitoring and analyzing phases, as well as between the planning and executing phases.

Figure 9 shows a general architecture of CPS, where two classes of devices may be highlighted: i) personal and environment devices, responsible for running services and applications, interacting with the user, external systems, and sensing/controlling the environment; and ii) sensors and actuators, responsible for obtaining information and acting upon the environment.

Sensors and actuators receive commands from the personal and environment devices and may be present in the environment (light switch, A/C) or present in the device itself (position, acceleration sensor). External systems interact with the architecture present in Figure 9 using available services present in environment and user devices.

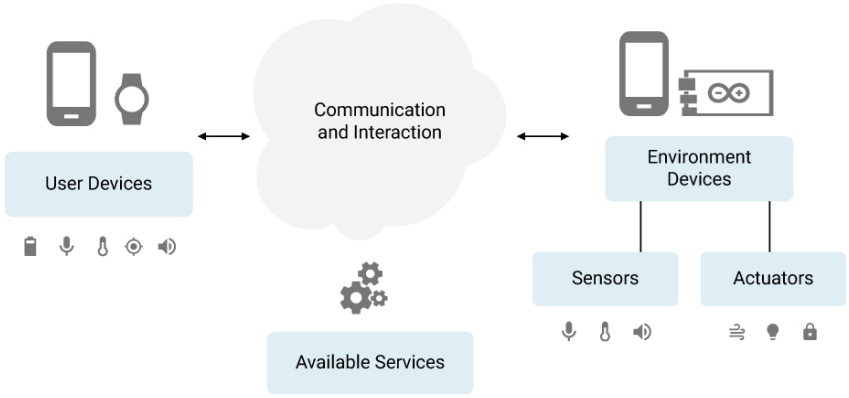

Figure 9 – Overview of CPS.

The interaction with the environment (personal and environment devices, sensors and actuators) is performed within two MAPE-K phases: Monitor and Execute. Monitor is responsible for sensing information about the execution environment, passing that information on to the Analyze phase. Execute is responsible for controlling the environment, by following actions generated in the Plan phase.

Another important requirement to implement decentralized self-adaptive CPS is the interaction and communication among user and environment devices. This interaction is necessary to implement the decentralized control patterns presented in section 2.3.

To better understand the requirements to develop CPS, a literature review on existing frameworks and middleware was conducted in this work. Initially, all papers published between 2010 and 2014 in the following conferences and journals were considered: *IEEE International Conference on Pervasive Computing and Communications (Percom), ACM International Joint Conference on Pervasive and Ubiquitous Computing (Ubicomp), Personal and Ubiquitous Computing, IEEE Pervasive, Pervasive and Mobile Computing*. The goal was not to consider all conferences and journals, but rather to understand the requirements for constructing CPS, from the perspective of prominent conferences and journals. Hence, 174 papers were found, their keywords were extracted to describe each requirement and organized according to the following requirements categories:

- Acquisition and diffusion of context information - 29 papers

Keywords extracted: *Gossip, information dissemination, Participatory sensing, distributed, sensors, context acquisition, indoor navigation, context distribution, context management, context provenance, quality of context, real-time, context monitoring, content sharing,context specification*

- Context models - 17 papers

  Keywords extracted: *User Situation, context-prediction, user preferences, activity recognition, context-models, context reasoning, behavior modeling, situation-aware*

- Specific application domain - 9 papers

  Keywords found: *Social networks, MANETS, Recommendation, Augmented Reality*

- Self-adaptation - 32 papers

  Keywords extracted: *Self-organization, Feedbacks loops, Self-adaptation, formal-specification, autonomic management, autonomic managers, goal-oriented, plan-based, Learning, multi-agents, Decision support, AI Planning, distributed policy resolution*

- Verification and validation - 5 papers

  Keywords extracted: *Testing, Formal verification, concurrent verification, assessment*

- Device interaction - 12 papers

  Keywords extracted: *Remote interaction, federation-based communication, device communication, opportunistic group communication, ambient interaction, Spontaneous interaction, channel models, seamless interaction*

- Programming models and languages, middleware and frameworks - 80 papers

  Keywords extracted: *Middleware, application development, Architectural patterns, design concerns, component, adaptation, reference model, requirements, abstract representations, tasks, services, system support, design challenges, framework, ambient programming, programming language, mobile workflows, events, tuples*

- Interoperability - 4 papers
- Mobility - 5 papers
- Security - 2 papers

These categories were created according to how the authors described them in their papers. It is important to note that the last three requirements (Interoperability, Mobility and Security) are keywords themselves and there is no need to further broken down into separate and simpler keywords.

Each paper has several keywords and may be present in more than one category. Ad-

ditionally, synonyms of keywords already present in the list were not inserted, but were counted. A spreadsheet containing the entire paper list is present at <http://www.great.ufc.br/index.php/o-great/quem-somos/professores-pesquisadores/37-marcio-espindola-freire-maia>. The amount of papers in each category serve as an indication of the category importance for the CPS/ubicomp/pervasive community.

Considering the support for developing and executing CPS, these papers were analyzed and divided into two groups: i) Communication and Interactions and ii) Managing sensors and actuators. These two groups were created to present an implementation view of all listed categories. That view will be used later on to present the contribution of this thesis.

2.4.1 Communication and Interaction

Communication and interaction is a basic requirement of decentralized CPS, accomplished by using interaction mechanisms. Figure 10 shows the exchanging messages, discussed in (MAIA *et al.*, 2009), using any available wireless technologies requirements for implementing communication and interaction in CPS.

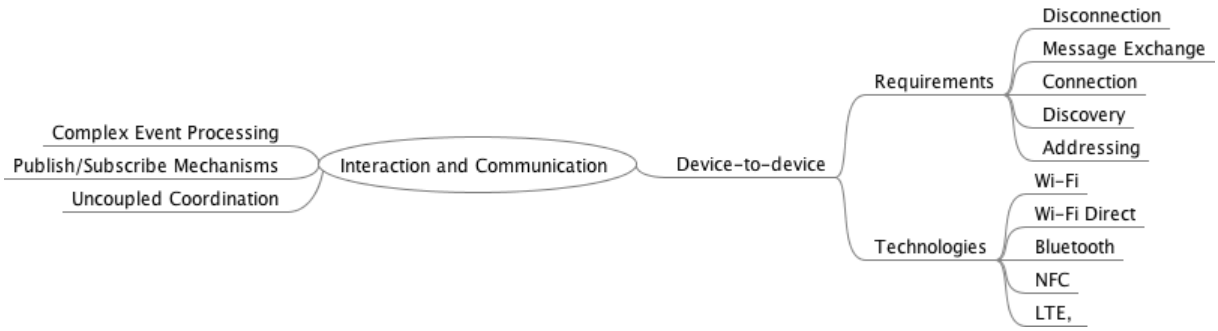

Figure 10 – Requirements for implementing communication and interaction.

From the wireless technologies perspective (Figure 10, device-to-device), there are several technologies to be used (Wi-Fi, Wi-Fi Direct, LTE, UMTS, Bluetooth, NFC) in CPS. Each communication technology has specific characteristics, considering their communication range, the existence of a fixed infrastructure, delivery guarantee and ordering. These characteristics make each technology suitable for a limited number of scenarios (MAKRIS *et al.*, 2013; PERERA *et al.*, 2014a).

Then, the usual approach is to define at development-time the communication technology (i.e. Bluetooth) and build the applications using it. That approach is too restrictive, because there is not one technology indicated to every situation (MAIA *et al.*, 2014). Opposingly,

self-adaptive CPS permits to postpone the decision about the communication technology to runtime. It permits also to change from one technology to another, in order to handle changes in the execution environment.

The second attribute is about interaction mechanisms. In order to deal with mobility and unpredictability, interaction should minimize the coupling among devices. Low coupling interaction based on tuple spaces and publish-subscribe have proven to be well-suited to CPS (MAIA *et al.*, 2009; LIMA *et al.*, 2011b).

CPS are usually composed of several independent subsystems interacting by message passing. One important characteristic of these systems is dynamicity. Thus, interaction between devices should be designed to minimize coupling. Uncoupling can be divided in three dimensions (EUGSTER *et al.*, 2003): i) time uncoupling permits that the message exchange process occurs between processes that are not available at the same time; ii) space uncoupling lets two processes to interact unaware of their physical address; and iii) synchrony uncoupling implements unblocking send and receive operations.

Two architectures to implement uncoupled interaction are used. The first is called event-based message exchange (HINZE *et al.*, 2009; MUHL *et al.*, 2015). It consists of one application subscribing to an event broker to receive events from an specific type. A publisher generates events and notifies the broker about the existence of new events. This broker is responsible to notify the subscribers when an event is generated. The second form of uncoupled in interactions is based on tuples spaces. Tuple-based mechanisms have a common tuple-space, where applications can read and write messages (tuples) in that space (CABRI *et al.*, 2006; MAMEI; ZAMBONELLI, 2009b).

2.4.2 Resource Management

The Monitor and Execute phases (Figure 2) are responsible for interacting with sensors and actuators present in the environment. Hence, to implement these two phases, programming environments should provide abstractions that support the development of CPS and hide away from developers the low-level complexities, such as dynamicity and uncertainty of resources, mobility and heterogeneity. To take advantage of high-level abstractions and minimize the programming effort, usually developers rely on a middleware layer to implement such applications. Figure 11 shows the requirements for implementing resource management in CPS (HOLLER *et al.*, 2014), which are detailed in the next subsections.

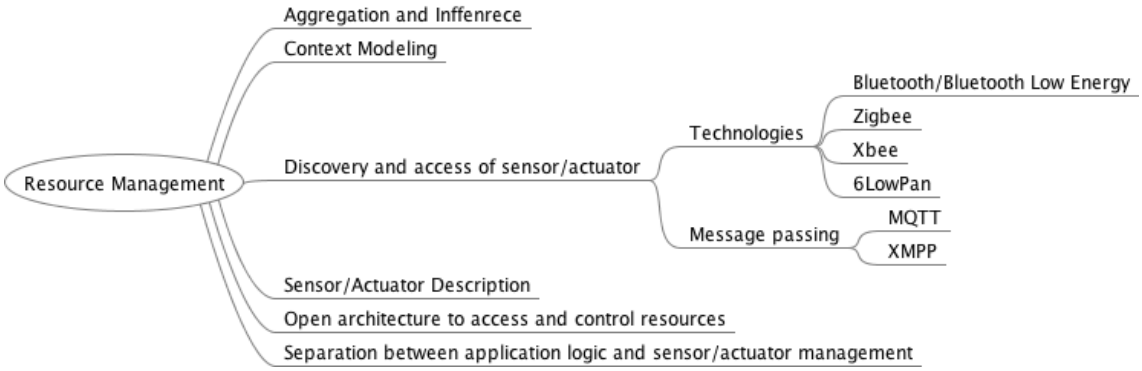

Figure 11 – Requirements for implementing resource management in CPS.

2.4.2.1 Separation between sensor/actuator and application logic

The first requirement is to permit the separation between sensor/actuator and application logic, since several CPS may execute concurrently in the user environment (LORENZ, 2013; MAYER *et al.*, 2014; BRANDAO *et al.*, 2013; VLIST *et al.*, 2013). Despite the distinct application domains (e-heath, smart home, etc.), they share common data and services (e.g., access to environment sensors data, actuator invocation). The separation of the business code from the code of the communication with sensors and actuators improves reuse.

To improve modularity, a component infrastructure to access sensor and actuator facilitates the development and management of each resource (BRANDAO *et al.*, 2013; MAUREL *et al.*, 2011; CHOUITEN *et al.*, 2011; HUGHES *et al.*, 2009; THOELEN *et al.*, 2014). This modularization should also help to reduce the coupling among applications, services, sensors and actuators, facilitating its adaptation. Since the environment is dynamic, the availability of sensors or actuators can change. Thus, less dependence on the application code from the code that controls sensor and actuators, means smaller complexity to reconfigure the application to address environment dynamicity.

2.4.2.2 Open architecture to access and control resources

CPS envisages a huge number of sensors and actuators deployed in the user environment. They need to use open standards in order to control, communicate and interact with these devices. The openness of a distributed system summarizes this design principle, which determines whether a system can be extended and modified in various ways (COULOURIS *et al.*, 2011).

Standardization of interfaces (including those of the sensors, actuators, and the

system itself) is a fundamental design principle to provide openness (LEE *et al.*, 2013; THOELEN *et al.*, 2014; SONG *et al.*, 2014; BORGIA, 2014; TRILLES *et al.*, 2015). The key software interfaces of the system components should be made available to software developers using Application Programming Interfaces (API) or services interfaces. Then, a software developer can incorporate new features, new sensors and new actuators on the system more easily.

2.4.2.3 Abstractions to describe, discover and access resources

According to the architecture present in Figure 9, the management of sensors and actuators takes place between personal or environment devices and sensors/actuators. One basic assumption in CPS is that interacting devices may move away, increasing the complexity to manage the entire system from a centralized manager.

Self-adaptive CPS assumes a large number of devices (sensors and actuators) that interact autonomously. Some objects will start communicating and others will disappear (PERERA *et al.*, 2014b). Hence, CPS development shall use a layer or infrastructure that manages and hides the complexity and dynamicity of the environment.

In traditional distributed systems, a service abstraction is used to encapsulate a collection of related resources, and the system presents their functionality to users and applications using a service interface. Services restrict resource access to well-defined set of operations. The provision of a similar concept in CPS to encapsulate sensors and actuators is a desirable design principle (CAPORUSCIO *et al.*, 2012; CASTELLI *et al.*, 2011; SASSI *et al.*, 2014; THOMA *et al.*, 2014; ZHANG *et al.*, 2014b). The idea is to uniquely identify reachable sensors and actuators, which are activated, and how the system should proceed to gain access to these resources.

This service and resource management can be implemented independent of an application or domain. Traditional discovery services are based on the triad: a service publication via a standard interface, a service repository, and a resource discovery service. A service provider (a sensor) will publish in a repository (distributed or centralized) or announce in the network its interface (available operations and data) and how it can be reached. A service client, such as an actuator or an external system, requests service providers to the repository or search in the network by using a service discovering protocol. Once the service is found, the client makes a connection or a direct invocation of the service provider. For example, the actuator accesses the value of the ambient temperature sensor. Similar resource discovery services are

implemented with different technologies (e.g. RMI, OSGi, W3C Web Services), and also using different middleware, such as GAIA and SOCCAM (ZHAO *et al.*, 2007). Service abstraction for representing sensors and actuators also allow them to be composed and accessed with different levels of aggregation (GUINARD *et al.*, 2010; KAO *et al.*, 2015).

2.5 Discussions on the Running Example

The GREat Tour application presented in Section 1.3 takes place through the interaction between personal and environment devices. The system execution is based on environments (e.g. rooms, buildings), and users/items associated to each environment. Thus, when the user enter a specific environment, his/her devices must query for existing items and other users. The initial implementation of the GREat Tour application did not adapt its behavior according to environment changes, rather it adapted the content being presented based on the current indoor location and also according to present users and available items.

One problem with the previous implementation is the programming paradigm used. Once the mechanisms used to implement its functionalities were defined and implemented (e.g. indoor location based on QR-Code, list of users in a environment based on a centralized web service), these mechanisms could not be changed any longer, unless the application is re-implemented. Thus, in order to permit the GREat Tour app to self-adapt, mechanisms to change the application behavior would have to be designed.

According to the MAPE-K loop, on the bottom layer, the application must obtain information from sensors. In the GREat Tour app, information such as indoor location, nearby users and available items are obtained from sensors. Although the functionality is defined at design-time (e.g. indoor location), its implementation must be able to be changed at runtime (e.g. from QR-Code-based to RFID-based).

With information obtained from sensors in the Monitor phase, the Analysis phase has two possible actions: i) use functionalities present in the Execute phase, when the information about the environment has associated response actions or ii) call on the Planning phase, if the detected information represents a system state not previously designed. To introduce self-adaptation in the GREat Tour app would require to generate a sequence of actions for each environment the user enters. Thus, changing the environment would require changing the associated sequence of actions. Alternatively, if an exceptional state is detected, one without associated actions, the Planning phase is called to generate actions to deal with that exception.

The Monitor and Execute phases interact with nearby devices and sensors/actuators, according to the Master-Slave pattern (section 2.3.3). Nearby devices interact among them using wireless communication technologies (section 2.4.2) and interact with sensors/actuators present in the environment (section 2.4.1).

2.6 Conclusions

This chapter presented an overview of self-adaptation for CPS, aiming primarily to list the requirements to develop decentralized self-adaptive CPS. Here, self-adaptation was considered from the architecture specification point of view, to improve modularity and, as a consequence, reusability.

Architecture-based self-adaptation permits the development of CPS systems dividing them into layers and modules. According to the MAPE-K loop, information from the environment is obtained in the Monitor module and passed on to the Analyze module, to verify if the system is in a desired state. Then, if adaptation is required, the Plan module, using system models present in the Knowledge module, generates a sequence of actions and passes it on to the Execute module, responsible for acting on the environment.

Decentralization was analyzed in the MAPE-K loop, to understand the challenges to implement each one of the four phases when several distributed subsystems must coordinate their actions to achieve a desired state. Thus, five decentralized MAPE-based patterns were presented and analyzed.

Finally, considering decentralization as mandatory, the requirements to develop CPS were presented and existing technologies analyzed, considering the requirements discussed in section 2.4.1, Communication and Interaction, and 2.4.2, Resource Management. All available implementation is the solution space of self-adaptive CPS. They are assessed at runtime to define the most appropriate solutions considering application goals and execution state.

The primary goal of this thesis is to provide an infrastructure to support the development of decentralized self-adaptive CPS. That infrastructure should help developers to implement the Monitor and Execute phases from the MAPE-K loop. Another goal is to implement communication and interaction mechanisms, with two design principles: uncoupled interaction among all system entities.

3 RELATED WORK

This chapter presents the related work divided according to the following layers: Communication, Coordination, Execution and Adaptations. This chapter also presents solutions that integrate these layers to offer support to self-adaptation in CPS.

3.1 Communication Layer

In order to create a communication infrastructure to handle the requirements of CPS, dealing with mobility and decentralization, it is mandatory to consider the following aspects: i) to provide means for applications to explore physical proximity; ii) to use available communication technologies; and iii) to permit the adaptation of the communication technology.

3.1.1 Exploring Proximity and Mobility of Devices

In cyber-physical systems, devices ought to exchange data and control signs without relying on any pre-established communication infrastructure. They explore physical proximity to communicate when a fixed infrastructure is unavailable or rely on other devices to carry-and-forward and multi-hop messages when direct exchange is not possible ((CONTI *et al.*, 2010)). In order to carry on this transparent communication paradigm, developers use technologies such as Bluetooth, Wi-Fi Direct, Wi-Fi or 3G/4G to implement the communication requirements of their applications.

Communication without the use of existing infrastructures has been the subject of study for quite some time now. Approaches like mobile ad-hoc networks (MANET) ((SARKAR; LOL, 2010)), delay-tolerant networks (DTN) ((WHITBECK; CONAN, 2010)) and opportunistic computing (OC) ((CONTI *et al.*, 2010)) all present concepts that can be used to implement ubiquitous computing.

According to (HUANG *et al.*, 2008), these approaches ought to consider the following restrictions:

1. **Nodes may be mobile.** Before executing any actions, applications must query the environment at runtime for reachable devices and resources.
2. **No assumption about the communication infrastructure should be made.** Approaches that use multiple technologies and adapt at runtime according to available technologies are preferable.

3. **No assumption about the network topology should be made.** Any kind of centralization should be avoided. If necessary, it should be coordinated at runtime by the devices themselves.
4. **Nodes may pass on messages intended for other nodes.** Multi-hop and carry-and-forward strategies are used to permit devices to communicate, even when they are not within each others' communication range.

Although MANET, DTN and OC have been studied for quite some time, their use is not thoroughly diffused in real-world applications. One reason is the complexity to handle these aforementioned restrictions. It requires from application developers to implement not only the functional requirements of the applications, but also to deal with intermittent connectivity, routing (multi-hop and carry-and-forward) or adaptation of the communication technology.

Another important problem faced by developers nowadays is to implement and deploy routing solutions on existing mobile devices. For instance, Android[1] and iOS[2], two popular operating systems of mobile devices, give very little support to system services such as routing. Thus, any solution that uses multi-hop or carry-and-forward strategies must implement it at the application layer. One alternative is to modify the existing implementation on the OS, but has limited applicability, since it requires from applications access to system resources that most users do not have.

To implement multi-hop and carry-and-forward routing on the application layer from scratch, the developer would use technologies such as Bluetooth and WiFi Direct, implement communication requirements such as neighbor discovery, connection, disconnection and consider adaptation to changing environment condition. Only then, he/she would implement the actual behavior of the routing algorithm.

3.1.2 Exploring Available Communication Technologies

Mobile devices nowadays are capable of communicating using several communication technologies, such as Bluetooth, Wi-Fi, Wi-Fi Direct, NFC or 3G/4G, used by application developers to create distributed applications. The usual approach is to define at development-time the communication technology (i.e. Bluetooth) and build the applications using it. That approach is too restrictive, because there is not one technology indicated to every situation.

[1] http://www.android.com/
[2] http://www.apple.com/ios/

One alternative is to implement applications that choose at runtime, among the available technologies, the most fitted communication technology at a given situation. Here, the problem is the complexity to manage more than one communication technology and to adapt it at runtime according to availability or non-functional requirements such as latency or throughput. To use more than one technology, developers would have to implement neighbor discovery, connection, disconnection, message exchanging in each technology, along with the integration between application and all technologies.

Allowing developers to explore multiple communication technologies is an important design decision of cyber-physical systems. It improves the availability of the overall system to deal with disconnections and mobility. Thus, an important contribution is to create a framework offering abstractions (interfaces) to hide away the complexity from developers. In this way, they are able to focus solely on application requirements.

3.1.3 Runtime Management of the Communication Layer

Developers access functionalities to use the communication technologies through common interfaces, like discovering available resources and sending messages. Additionally, these interfaces are also used to notify the application of events occurring in the network, such as connection of new devices or unavailability of the entire communication technology.

Although it might be important to hide from developers the implementation of technology-specific actions, to improve simplicity, this approach has the disadvantage of hiding information about the execution environment. In that direction, since cyber-physical systems usually face volatile environments, low-level information is generally used to adapt system behavior. Thus, technology-specific actions should be kept hidden from developers and accessed through common interfaces (for simplicity). However, information about the execution environment should be made available, and management interfaces provided to handle environment changes (for flexibility).

Considering the communication technologies available (e.g. NFC, Bluetooth Low Energy, Bluetooth, Wi-Fi Direct, Wi-Fi, UMTS, LTE), they have static information that describes their behavior, such as nominal bandwidth, communication range or security, as well as dynamic information that can only be accessed at runtime, such as latency, packet loss or available devices. Thus, depending on application requirements and runtime information, each technology is tailored to an specific scenario.

A framework that handles communication in cyber-physical systems should offer mechanisms to access information about the available technologies and their execution information, plus management facilities to let developers indicate which technology is to be used in a given scenario.

On the technology level, a communication technology may be described statically, according to its nominal speed, range, security protocols, or dynamically, according to its latency of packet loss. Adaptation at this level chooses the technology according to its static and dynamic information, along with application requirements.

Additionally, according to (MAIA *et al.*, 2009), the routing algorithm, implemented using the available technologies, is tailored to specific situations, like centralized or distributed communication, mobility range or number of neighbors. Here, adaptation can be accomplished changing the routing algorithm or adapting message dissemination parameters. Adaptation at this level ought to provide information about the execution environment, such as number of neighbors, message loss, mobility patterns, along with mechanisms to adapt and change the routing algorithm.

3.1.4 Related Work on CPS Communication

(DVINSKY; FRIEDMAN, 2014) proposes a multi-platform group communication framework targeted at mobile devices called Chameleon, built based on a layered architectural model, where each layer is responsible for a particular communication requirement (for instance, routing, delivery guarantee, ordering, addressing). These layers are linked using event chains (similar to a pipeline). Using this design, when increasing the number of layers and its event chain, the complexity to adapt the structure of each layer increases accordingly, since the interfaces between layers are immutable.

Multiple communication technologies in Chameleon are present through the implementations of TCP and UCP on the transport layer, configured prior to execution. The downside is this approach is that communication is based on IP networks and all communicating devices must be accessible by their IP addresses. That is not possible, for instance when cellular networks are being used, or to communicate directly using Bluetooth and Wi-Fi.

(LI *et al.*, 2014) proposes CA-P2P, a network layer architecture for the construction of mobile peer-to-peer applications, exploring both direct and infrastructure-based communication. CA-P2P is built based on the premise of using direct communication when possible,

bringing benefits such as improved reliability, energy consumption and offloading from the core communication infrastructure. Another premise of CA-P2P is the use of multiple-technology. Communication among peers is managed by virtual leaders, which are special peers responsible for a group of peers. CA-P2P can be used as underlying architecture for the implementation of multi-hop and carry-and-forward algorithms (MAIA *et al.*, 2014). Additionally, they do not define the communication technology to be used.

AllJoyn (ALLJOYN, 2013) offers an infrastructure that handles the underlying communication requirements such as neighbor discovery, connection and message exchanging. Aiming to provide an infrastructure for the Internet of Things, AllJoyn is multiplaform and has implementations for Android, iOS and Arduino. The current implementation of AllJoyn lacks support for multi-hop and carry-and-forward. Developers would have to implement them on top of what is offered by AllJoyn.

The Haggle Project described by (CONTI *et al.*, 2010) created an infrastructure to develop applications based on the opportunistic computing paradigm. Its provides to developers primitives that handle trust, security, resource discovery, and carry-and-forward routing capabilities. Similar to the work proposed here, it permits application developers to use different communication technologies. It does not have implementations of multi-hop algorithms.

AmbientTalk/M (SUZUKI *et al.*, 2011) is an object-oriented programming language aimed at peer-to-peer mobile applications. It offers to developers implementations of multi-hop algorithms, possibly among multiple networks. It does not offer implementation of carry-and-forward algorithms.

Table 1 – Characteristics of the related work

	Multi- technology	Multi- platform	Multi- hop	Carry- and- forward
Chameleon	✓	✗	✓	✗
CA-P2P	✓	✗	✗	✗
AllJoyn	✓	✓	✗	✗
Haggle	✓	✗	✗	✓
AmbientTalk	✓	✗	✓	✗

Table 1 presents a summary of the related work presented in the communication layer. The characteristics analyzed focused only on the communication requirements for cyber-physical systems.

3.2 Coordination Layer

Because of the volatility and dynamicity of CPS, direct interaction should be minimized. Thus, using approaches based on shared spaces and events (Refer to section 2.4.1) is important.

A tuple-based approach was initially proposed to be used by the Linda programming language (CARRIERO; GELERNTER, 1989) to foster uncoupling among interacting entities. A tuple is an abstraction within the Linda programming model to represent any data exchanged among system entities, using a key-value pair. The use of tuple spaces improves uncoupling, by separating the interacting entities, and permits the creation of systems that are more flexible to maintain and evolve, since tuples have no schema, and the data format is agreed between interacting entities at runtime. A broader discussion on the benefits of using tuples is presented in Appendix A.

Distributed entities interact using the tuple space by reading and writing information in the shared space. Applications generating data (value) write it to the shared space using a description to that data (key). To read information from the shared space, applications must proactively access the space informing the specific key they are interested in.

On top of the tuple-space, an event mechanism can be used. Here, instead of proactivelly accessing the shared space, applications can be notified when tuples with specific characteristics are written onto the shared space. Several works propose solutions integrating the tuple space model and the event model. (MURPHY; PICCO, 2006; CURINO *et al.*, 2005; MAMEI; ZAMBONELLI, 2009a; NETO *et al.*, 2013b).

The related work listed in this section presents coordination models adapted from the Linda model and are all based on distributed tuple spaces. They introduce the idea of reaction, based on event mechanisms, reacting to changes in the tuple space. They were also projected to mobile and context-aware systems, and support operations in ad hoc networks, thus adapting to the main requirements of ubiquitous computing.

Lime (Linda in a Mobile Environment) (MURPHY; PICCO, 2006) is a middleware that enhances Linda model, allowing it to be used in mobile environments. It adopts an approach of decentralized tuple spaces, sharing the tuples according to the connectivity of its mobile components. Thus, the tuple space does not remain associated to a specific device. Each process has it own tuple space, and this one is shared with the remaining local processes or it can be accessed through a communication network. The access to remote tuple spaces is done in a

transparent way, whereby the tuple reading operation is distributed, as writing operation is local.

TOTA (Tuples On The Air) (MAMEI; ZAMBONELLI, 2009b) is a middleware focused on decoupled communication between the distributed application components. A TOTA version is embedded in each device and the tuples are propagated through the network according to propagation rules available in the tuple itself. Different from Lime, the reading operation in the tuple space is made locally, and the writing operation is distributed. TOTA constantly monitors the network composition, providing information about who enters and leaves the network.

EXEHDA-TS (SOUZA *et al.*, 2012) is a scalable coordination model with dynamic behavior. It manages tuples distributed in tuple spaces available in each system node. It also implements mechanisms based on events that notify the applications about relevant information as they are inserted in the tuple spaces. The system scalability is ensured by visibility parameters that restrict the access to the reading of distributed tuples.

3.3 Resource Management

The development and execution management of CPS to access and control environment resources is based on the description, discovery and access to these resources. On top of that, mechanisms to model, aggregate and infer high-level context have been proposed.

3.3.1 Resource Description, Discovery and Access

Most of the abstractions to describe, discover and access resources in CPS fall in one of the following approaches: i) services and ii) components.

3.3.1.1 Service-based Approaches

The use of services in mobile, ubiquitous and pervasive applications has be extensively studied (RAMBOLD *et al.*, 2009; MAIA *et al.*, 2009; ISSARNY *et al.*, 2011), and more recently for CPS (HOANG *et al.*, 2012). In that direction, ubiSOAP (CAPORUSCIO *et al.*, 2012) is a service-oriented middleware for accessing resources in ubiquitous environments, divided in two layers: i) network-agnostic connectivity, to permit the interaction with services to take place independently of the communication technology; and ii) WS-oriented communication, to levarage the functionalities offered by the WS-* specification.

The network-agnostic connectivity offers functionalities to permit the interaction

using several existing communication technologies. Thus, it implements an addressing scheme on top of each technology and permits the network selection according to QoS parameters. Finally, message exchange can be carried out using unicast and broadcast.

Additionally, the WS-oriented communication uses a description language called the ubiSD-L, which is a description language to permit the publication and discovery of services, similarly to regular web services. The service-based approach proposed by ubiSOAP is an attempt to permit environment resorces to be accessible over the Internet. Another benefit is a common mechanism to describe resources (services) with an improved interoperability (as long as all interacting entities use ubiSD-L). The downside is the use of SOAP messages, which has been replaced recently by REST-based approaches (for lightness) (CAPORUSCIO *et al.*, 2014).

Most approaches exploring service-orientation use a middleware layer to implement basic requirements, such as service description and discovery, message exchange, coordination and security (MAIA *et al.*, 2009; HOANG *et al.*, 2012). The downside here is the requirement of all interacting entities to use the same middleware layer. To mitigate this problem, the concept of emergent middleware has been proposed (BROMBERG *et al.*, 2011; BLAIR *et al.*, 2011), where each interacting module can use its own middleware and the interaction is based on a mediator, automatically generated at runtime according to the middlewares used by each module.

The mediator is generated using the STARLINK framework (BROMBERG *et al.*, 2011), based on a formal model to describe the message type and sequence, along with the behavior of each individual middleware. A mediator is only generated to permit the interaction of two different middlewares if they present the same behavior and the messages types and sequence sent by one middewlare is compatible with the types and sequence the second middleware is expecting to receive.

Another approach to improve the flexibility in service-oriented ubiquitous applications is AMESMA (MAIA *et al.*, 2012), where essential services such as description and discovery are accessed using common interfaces exposed by an API. The linking between interfaces and the actual implementation is only defined at runtime using information received by the execution environment.

Service orientation was initially proposed to separate the interacting entities (service producers and consumers) to promote uncoupling. That separation was used to develop mobile/ubiquitous/CPS applications to deal with disconnections, mobility, unpredictability and interoperability. Although the use of services was an evolution considering requirements in

CPS, there were several challenges that still remain to be considered, such as heterogeneity and resource-scarcity.

3.3.1.2 Component-based Approaches

Component-based development (CBD) proposes the construction of systems based on existing software units called components. Hence, instead of one monolithic system, a software is a collection of interacting units (LAU; WANG, 2007). A component model describes how to build, deploy and use components using a component description language, which is a formal or semi-formal notation to specify components based on offered and required interfaces, pre- and pos-conditions and invarations.

In CPS, adaptation is a mandatory requirement, and the component composition at runtime permits the system to use components according to environment information and specific characteristics of each component.This is possible because each component specifies its non-functional execution and composition characteristics (CRNKOVIC *et al.*, 2004; HUANG *et al.*, 2009), using notations to describe the system architecture called Architecture Description Language (ADL) (GARLAN *et al.*, 2000).

An ADL permits the description of complex systems based on component interactions. Thus, functional requirements are delivered using specific components and non-functional ones are guaranteed by analyzing the component relations at runtime. This dichotomy permits the system execution based on components, and the analysis of its execution properties based on system models constructed using ADLs.

This approach is used by **Kevoree**, which is a component model based on the models@runtime (BLAIR *et al.*, 2009), to support runtime system adaptation. Models@runtime uses reflection, which is the system capability to self-inspect and analyze execution properties. Additionally, a runtime architecture is used to implement application functional requirements.

Kevoree uses four basic units: i) Components, which are processing units; ii) *Node*, representing system devices; iii) *Group*, to model the communication among components and iv) *Channel*, to implement the communication semantic among Nodes.

Adaptation in Kevoree is specified using a script-based language called KevSript, used to implement adaptation policies according to runtime infromation. Self-adaptation in Kevoree is present using a formalism based on control loops(KRAMER; MAGEE, 2007).

Another component model is called **DEECo** (BURES *et al.*, 2013). The main entity

in DEECo is an *Ensemble*, which is a group of components interacting to achieve a common goal. An *Ensemble* is the only possible interaction model, and it defines at runtime the participating components. The communication management within an Ensemble is defined by a coordinator, reponsible for defining who and when can receive a message.

Components in DEECo possess i) knowledge, which defines the component state based on exposed interfaces; and ii) processes, which are the behavior to change the system knowledge. Communication among components (processes) uses Ensembles.

While Kevoree permits the specification and verification of system properties using its architecture, it makes it difficult to guarantee these properties in decentralized systems. That is the main advantage of DEECo, since it uses a component model based on interactions using ensemblers.

An important initiative to promote reuse and minimize coupling is the ***Open Services Gateway Initiative - OSGi*** (ALLIANCE, 2003), composed of open specifications for the management of services and components in distributed applications. Using OSGi, every component or service is a *Bundle*. Additionally, a centralized *Gateway* is responsible for discovering, deploying, starting and stoping a *Bundle*.

OSGi offers four important functionalities for CPS developers: i) low coupling, by the use of components and services; ii) late biding, by separating the component/service interface and their implementations; iii) resiliency, since failures are dealt with by changing their implementation; and iv) location transparency, where every communication between application and component/service is managed by OSGi. In CPS, the *Bundle* life-cycle management is an important feature implemented by OSGi. The downside is the use of centralized *Gateway*.

Using an OSGi implementation, **iPOJO** (ESCOFFIER *et al.*, 2007) uses the Service-Component Architecture (SCA), where a component is deployed as a service and publishes its interfaces in a registry available for other applications. The principle behind *iPOJO* is that developers should be responsible only for implementing the application business logic, and component lifecycle is handled by iPOJO.

Using the SCA, the fraSCAti middleware is divided in three layers (SEINTURIER *et al.*, 2012): i) execution platform, formed by components exposing its execution and management interfaces; ii) non-funtional services, creating a mechanism for reusing implementations for non-funtional requirements, permiting the association between components and non-funcional services; and iii) business logic, which is the mechanism for discovering and invoking frasSCAti

components.

3.3.2 Context Modeling, Aggregation and Inference

ContextDroid (WISSEN *et al.*, 2010) is an Android-based context acquisition layer whose main benefits are its efficiency, extensibility and portability. Although it does not adapt its context acquisition layer, ContextDroid is built upon a component-based framework that uses the Android platform itself to perform component management.

Kramer et al. (KRAMER *et al.*, 2011) present an infrastructure to implement context acquisition, that monitors context changes independently of application. Their component model is focused on composition of sensor data.

Preuveneers and Berbers propose a middleware for mobile devices to tackle the all-and-nothing problem (PREUVENEERS; BERBERS, 2007), which is the restriction of which a system can only be installed with all its components, or not installed at all. Instead, Preuveneers and Berbers implemented using Java a mobile platform to decrease the number of sensors in execution, trying to reduce battery consumption. Low coupling among applications and middleware were not treated in the Preuveneers and Berbers approach.

3.4 Self-Adaptive Approaches

Zhang et al. (ZHANG *et al.*, 2014a) propose a three-layer self-management approach called *LinkSmart*. In the lower layer, they use service-orientation to foster interoperability among distributed entities, access to sensors and actuators based on reflection, to permit runtime introspection. Additionally, events are used in the communication between layers.

There are three types of sensors: i) state, to monitor information about service execution; ii) communication, to monitor information about the communication between services; and iii) configuration, providing information about system configuration and execution. On the other hand, actuators provide means to change architectural configurations using an architecture description language and a scripting language, responsible for reconfiguring the system by changing components.

The management layer responds to events on the component layer, changing the scripting to make the system enter a desired state, using OWL ontologies and SWRL *If-Then* rules [3].

[3] http://www.w3.org/Submission/SWRL/

Interoperability is dealt with in two different levels: in the component layer, by using service oriented architecture for the interaction among distributed entities, and in the management layer, by using ontologies. Scalability has become a problem in large systems, since inference using ontologies is resource-intensive.

Uncoupling is another important characteristic in CPS not considered by LinkSmart, since interactions occur based on direct service invocation. In that direction, they improve the flexibility by changing communication components, but are still based on direct interaction.

3PC (HANDTE *et al.*, 2012) presents a supporting infrastructure for ubiquitous systems, aiming to facilitate the runtime adaptation of the communication, supporting system and application. Using 3PC, every device is called Peer. Additionally, they create the Smart Peer Group concept, in which devices and applications participating in a group share their resources. Devices may enter and leave a group dynamically.

In the network layer, communication adaptation is managed by the BASE middleware, which is an object-oriented communication module to manage peer interaction based on the microbroker pattern (BUSCHMANN *et al.*, 1996), aiming to to offer uncoupled interaction on top of a remote method invocation.

Interaction is accomplished using plug ins, which can be changed at runtime, according to environment information and plug in characteristics. The configuration is accomplished based on interfaces exposed to the upper layers. Developers use these interfaces to manage the interaction among peers by choosing the most fitted plug in.

In the system support layer, 3PC uses SANDMAN (SCHIELE, 2007) to adapt services offered within peer groups, managing available services to handle device disconnection. The management of available services uses a *Service Discovery System Service*, which uses a mediator acting as a catalog of available services and clients requirements. The mediator is elected among available peers in a peer group.

In the application layer, PCOM (BECKER *et al.*, 2004) uses component description to specify application functionality, but only at runtime the actual implementation is defined, according to available services.

Interoperability is considered by using service orientation, accessed based on the mediator. That is an interesting concept to implement on-the-fly connectivity and is useful in small to medium systems. Large systems may suffer efficiency problems, because the mediator centralizes the service discovery process, and the election algorithm introduces overhead on the

network and devices.

The fraSCAti middleware presented by Seinturier et. al (SEINTURIER *et al.*, 2012) uses the Service-Component Architecture (SCA) to implement dynamic reconfiguration in pervasive environments, both in application and execution infrastructure. fraSCAti is based on four specifications: i) assembler language, to specify how components are related at runtime; ii) component implementation, to specify component how each component is implemented; iii) linkage language, to specify provided and required interfaces for each component; iv) policies, to describe non-functional requirements offered by components and required by applications.

fraSCAti is divided in three layers: execution platform, formed by components providing interfaces to invoke components and interfaces to monitor and control them, along with a reflection mechanism to access runtime information about the systems; ii) non-functional services, to implement and reuse non-functional requirements, and a mechanism to associate components to non-functional services; iii) business logic, to implement description and discovery of fraSCAti services.

The execution layer is based on the FRACTAL component model (BRUNETON *et al.*, 2006), created to permit the implementation of reconfigurable systems, independent from programming language. FRACTAL permits to describe behavior and structure using a description language.

A concept called Persona is used to implement specific policies for each component, permitting the specification of non-functional requirements, along with their runtime interaction and how each component is instantiated, started, binded to other components and how the structure can be reconfigured.

Runtime structure is described using a interface description language and managed using the assembler language. Reflection is used to assess runtime properties. This approach permits reconfiguration of structure and behavior by adding and removing architectural entities according to environment information.

fraSCAti is a configurable middleware based on components and services. Self-adaptation is achieved by separating the managed and manager systems. Thus, the managed system is implemented using components, services and their bindings, plus the specification of properties of structure and behavior. These properties and specification is used by the managing system to inspect the managed system and adapt it when necessary.

Because each component is annotated with its properties, fraSCAti permits the

creation of decentralized systems, since inference over system properties can be executed locally at each device, improving scalability. Additionally, the interaction (connector) between system components can be replace at runtime, improving uncoupling.

The MUSIC framework (FLOCH *et al.*, 2013) provides a programming environment to facilitate the development of applications running in open and dynamic environment. It offers description and discovery, along with dynamic binding, of components, services and environment sensors.

Self-adaptation is accomplished based on an adaptation plan, permitting to applications to handle environment changes. The plan includes descriptions to local components and remote services, using a MAPE-K-based control loop with the following input: i) execution model, including global goals, policies and restrictions; ii) available services and components; iii) runtime environment information.

The MUSIC architecture, acting as managing system, is divided in four modules: i) Kernel, implementing basic functionalities to deploy, start, stop and undeploy components, acting as the basic execution unit in each device; ii) System services, providing access to remote services and execution state; iii) Context middleware, monitoring and notifying interested devices about context changes; iv) Adaptation middleware, altering the managing system using the MAPE-K control loop.

Although the discovery of services is based on shared spaces for uncoupling, interaction is based on direct invocation of services. A centralized control loop is also another concern, when large or dynamic systems come into place.

CAMPUS (WEI; CHAN, 2013) is a middleware to permit the automation of adaptation decisions by the middleware layer at runtime, based on two key concepts: i) ontologies to specify environment and system states and ii) description logic/first-order logic reasoning to define to infer adaptation policies automatically at runtime.

The architecture presented by CAMPUS is divided in three layers: i) programming layer; ii) knowledge layer; and iii) decision layer.

The programming layer is responsible for implementing and reconfiguring applications according to adaptation actions received from the decision layer. These actions are responsible for adding, removing and replacing parts of the applications. In this layer, CAMPUS permits the separation between adaptation and application functionality relying on Services, as an abstraction to represent an application process (sequence of functionalities). A process is a

sequence of simpler operations called Tasks.

The knowledge layer captures information about the environment, such as execution information required by services, non-functional restrictions imposed by applications and context data for adaptation decisions. All that information is described by a knowledge model presented by CAMPUS using ontologies.

Adaptation actions, if required, are generated in the decision layer, by changing the Task implementation. These actions are defined by analyzing the ontologies instances to select possible Task adaptation alternatives and instances describing the context information affecting the current Tasks. From the available remaining Tasks, the best Task is selected according to the existing adaptation rules.

Although the general architecture presented by CAMPUS is tailored to implement self-adaptation, the use of ontologies would restrict its use in CPS. Inference using ontologies requires processing capapabilities, which may not always be available. In large systems, ontology inference is not the most indicated solution. Additionally, CAMPUS does not consider uncoupling in its design.

Decentralized self-adaptation is dealt with in (JIAO; SUN, 2013). Authors sought the implementation of control loops for decentralized dynamic and volatile systems. They present the concept of Autonomous Component - AC, which are self-managed software entities capable of making local decisions based on behavior policies and satisfaction metrics.

Behavior policies are constantly assessed to identify which actions led to better satisfaction metrics. When a situation repeats itself, efficient actions are executed again.

This approach improves coupling among entities, making scalability more attainable. However, total decentralization increases the complexity to manage and guarantee overall system properties. This happens because it is not always straightforward to specify system goals from local individual behavior, where each AC makes local decisions based on partial view of the system. Another problem is to manage and control access to shared resources.

Table 2 – Characteristics of the related work on self-adaptation

	Interoperability	Scalability	Extensibility	Planning
Link Smart	✓	✗	✓	✓
3PC	✓	✗	✓	✓
fraSCAti	✓	✓	✓	✗
MUSIC	✓	✗	✓	✓
Jiao 2013	✗	✓	✓	✓
CAMPUS	✓	✗	✓	✓

Table 2 shows the related work analyzing interoperability, scalability, extensibility and planning. Interoperability is considered in most of the related work (except (JIAO; SUN, 2013)) by using service-orientation and the use of ontologies. While the use of ontologies has been use to solve the interoperability challenge, it restricts system scalability, in terms of number of interacting devices and time constraints.

Table 3 presents an analysis of the related work on self-adaptation, considering the requirements for building CPS, according to section 2.4.

Table 3 – Characteristics of the related work considering CPS requirements

	Commu-cation	Coordi-nation	Sensor/App Separation	Openness	Abstractions
Link Smart	✗	✗	✓	✓	✓
3PC	✓	✗	✓	✓	✓
fraSCAti	✗	✗	✓	✓	✗
MUSIC	✗	✗	✓	✓	✓
Jiao 2013	✗	✗	✓	✗	✓
CAMPUS	✗	✗	✓	✓	✓

Among the related work in Table 3, only 3PC considers the communication requirement. The remaining work assume the communication based on IP networks. *LinkSmart* has a mechanism to monitor communication state among services, but gives no support to multiple technology communication.

Uncoupled coordination is not fully considered by any of the related work. MUSIC uses shared spaces, but only as a service directory. MUSIC and fraSCAti accomplish uncoupling by using connectors that can be changed at runtime. Although it fosters uncoupling, it postpones to runtime the decision of which connector to be used.

3PC uses the concepts of microbroker, which permits the interaction mechanisms to be replaced according to environment information. However, that interaction is based on service invocations, coupling the interacting entities, at least, to be online at the same time (refer to Appendix A).

The separation between sensors/actuators and application is present in all of the analyzed related work, since they all are based either on services or components. For the same reason, abstractions to describe, discover and access resources are present in all related work.

Openness is a requirement to permit different developers to take part on the development/execution of a CPS. Similarly as the interoperability dimension on Table 2, only Jiao 2013 requires developers to use the same middleware layer.

3.5 Conclusions

This chapter presented the related work related to the topics considered by this doctoral thesis, dividing them into communication, coordination, execution and self-adaptation. The goal was to analyze each layer individually, to highlight the open issues in the area and to permit the creation of the contributions presented by this doctoral thesis.

On the communication layer, although the related work all support multiple technologies, they lack the possibility to develop and customize the routing strategies they use. On the coordination layer, existing tuple space-based solutions give support for distributed spaces, but the mechanism to exchange tuples with nearby devices is too brittle, with important adaptation restrictions.

On the execution and adaptation layer, the main problem presented by the related work is the coupling among interacting entities. Although existing work presented a few initiatives to improve uncoupling, they lacked full support for uncoupled coordination among distributed devices, sensors and actuators.

4 SUPPORTING DECENTRALIZED SELF-ADAPTIVE CPS

This chapter presents the main contributions of this doctoral thesis, which is an infrastructure to support the development and execution of decentralized self-adaptive CPS called CyberSupport. The goal is to offer to developers a software stack to deal with the Monitor and Execute phases in a decentralized MAPE-K loop: access and control of environment resources; message exchange and coordination; and application development and adaptation.

4.1 Introduction

In order to meet the challenging requirements of cyber-physical systems, self-adaptation permits the systems to reason about their environment and goals and change their behavior to handle changes and failures in order to maintain a consistent state (WEYNS *et al.*, 2013). Self-adaptation is implemented dividing the system in two parts (WEYNS *et al.*, 2013): i) the managed system, comprised of the interacting components; and ii) the managing system, which senses the execution environment and has the knowledge to take actions to adapt the system.

The system-wide support for the development and execution of uncoupled and decentralized self-adaptive cyber-physical systems, called CyberSupport is the primary contribution of this thesis, both from the perspective of the managed and the managing systems. The goal is to offer to developers a software stack to deal with the Monitor and Execute phases in a decentralized MAPE-K loop: access and control of environment resources; message exchange and coordination; and application development and adaptation.

Its main results are as follows: i) It provides mechanisms for the entities that form the system to interact transparently, both locally and globally, independently of the communication technology, presented in section 4.2.1 and published in (MAIA *et al.*, 2014); ii) It offers a mechanism that minimizes the coupling between all entities, facilitating the adaptation of the entire system, presented in section 4.2.2 and published in (NETO *et al.*, 2013b); iii) It is based on a component infrastructure to sense system information and to act upon system resources, presented in section 4.3 and published in (MAIA *et al.*, 2013).

Figure 12 shows the overall architecture of CyberSupport, divided in two layers; i) Communication and Coordination, responsible for exchanging and routing messages and the uncoupled coordination between the interacting devices; and ii) Execution and Adaptation, that

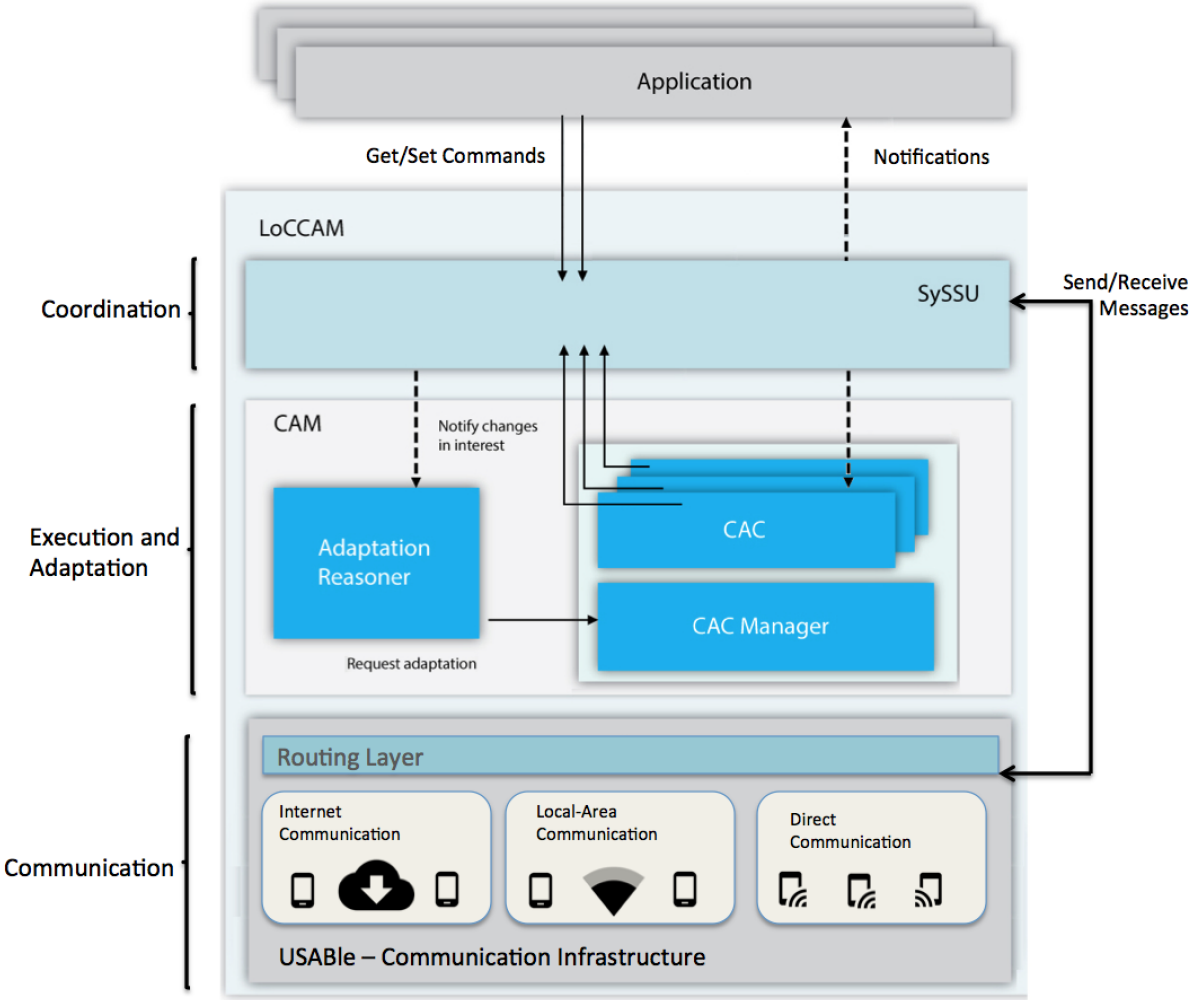

Figure 12 – Architecture of CyberSupport

provides a component model to permit the flexible construction of these applications.

The Communication and Coordination layers present two main results; i) A communication framework called USAble for the development of communication primitives for CPS that are independent of communication technology; and ii) USABle, a distributed coordination middleware based on tuples to foster uncoupled coordination.

USABle (MAIA *et al.*, 2014) is a communication framework to facilitate the creation of decentralized communication requirements that explore device mobility and proximity in a technology independent manner. Its main results are two-fold, as follows:

First, it offers to developers a set of technology-independent interfaces to handle communication requirements such as neighbor discovery, connection and disconnection, as well as message delivery. To developers of CPS, another result is the development of two well-known routing algorithms that can handle multi-hop and three implementations of carry-and-forward strategies.

Second, it is based on a modular and extensible, layer-based, architecture that permits developers to plug in new communication technologies and routing strategies, reusing much of the existing functionalities.

SysSU-DTS (NETO *et al.*, 2013a)[1] extends SysSU[2] (System Support for Ubiquity) (LIMA *et al.*, 2011a) and is a middleware to support coordination in ubiquitous systems. The benefits of SysSU-DTS are its expressibility and interoperability, since it provides a flexible way to access, filter and aggregate tuples within a tuple space. SysSU-DTS is built using USAble, permitting the coordination among decentralized devices using any tuple space present in nearby devices, as well as geographically dispersed ones.

On the Execution and Adaptation layer, the sensor and actuator component model is implemented using the LoCCAM middleware (MAIA *et al.*, 2013). LoCCAM (Loosely Coupled Context Acquisition Middleware) is a sensor and actuator management middleware based on three modules: 1) Life-cycle management of Sensor/Actuator components; 2) A coordination mechanism based on tuple spaces (SysSU-DTS); and 3) The self-adaptation and life cycle manager.

The vocabulary available to represent component information is organized as an information tree (Context Information Base - CIB). It can be referenced by a sequence of node names separated by dots. A similar mechanism is found in the network management domain called Management Information Base (MIB)[3]. For instance, the temperature in a given ambient can be accessed as 'context.ambient.temperature'. The tree offers the keys to access sensor information and to control actuators using the tuple space and plays an important role in the self-adaptation process.

This approach helps to uniquely identify the type of context information being described, since entries in this tree are available to developers to specify the information an application is interested in, and to LoCCAM, in order to manage the components required by applications at runtime.

4.2 Communication and Coordination Layers

Users carry a plethora of computing-enabled devices, such as smartphones, tablets, smartwatches and glasses, equipped with a large number of sensors and communication capabili-

[1] Work of a master student co-advised by myself
[2] Available in http://code.google.com/p/syssu/
[3] http://www.ieee802.org/1/pages/MIBS.html

ties. Not only users carry around personal devices, but the physical environment in which users are present is becoming filled with sensors and controllers as well.

Mobile devices interact among themselves and with fixed resources using wireless technologies. The interaction is comprised of two dimensions, namely communication and coordination. The former is implemented using available wireless technologies to exchange data (e.g. Wi-Fi, Wi-Fi Direct, LTE, UMTS, Bluetooth, NFC), while the later is the capacity of devices to execute a given functionality by coordinating their actions by exchanging messages.

The next subsections present the communication and coordination layers of the architecture presented in figure 12.

4.2.1 Adaptive Communication Framework for CPS

Although the number of communication technologies is vast, building a cyber-physical application from scratch considering communication requirements, such as device discovery and addressing, device connection and disconnection, as well as message exchange, it is not a trivial task.

The challenges lie on the distributed, volatile, mobile and large scale characteristics of CPS. Thus, to create applications that explore environment resources, system designers would have to choose the communication technology to be used, based on application requirements and restrictions, such as user mobility or communication range. For instance, applications exploring device proximity, such as the opportunistic computing paradigm (CONTI *et al.*, 2010), should use Bluetooth, Wi-Fi Direct or NFC. On the other hand, for applications with high mobility requirements, like transportation systems (BRAGA *et al.*, 2012), communication based on cellular networks (LTE, UMTS) are an alternative. The situation becomes even more challenging for applications interacting with both local and geographically dispersed resources.

According to (MAIA *et al.*, 2009), having decided the technology to be used, system designers must handle communication requirements such as resource discovery, addressing, message exchange, device disconnection, mobility and routing. To satisfy these requirements, two approaches are possible: 1) implement from scratch all requirements, which usually is complex and costly; or 2) use an existing middleware infrastructure, what may be too brittle (BLAIR *et al.*, 2011).

The middleware infrastructure should be flexible enough to change its structure and behavior, according to changes in the execution environment (LIN *et al.*, 2011), without relying

on any pre-established communication infrastructure. It should explore physical proximity to communicate when a fixed infrastructure is unavailable or rely on other devices to carry-and-forward and multi-hop messages when direct exchange is not possible (CONTI *et al.*, 2010). In order to implement this transparent communication paradigm, developers use technologies such as Bluetooth, Wi-Fi Direct, Wi-Fi or 3G/4G to implement the communication requirements of their applications.

To aid developers in handling communication requirements, this doctoral thesis presents USABle, which is a modular, extensible and adaptable communication framework for Cyber-Physical systems, based on the following principles:

I) Communication technology flexibility, where each technology is implemented based on pluggable components, that can be instantiated, started and stopped at runtime. Thus, separating application and communication logic, facilitating the adaptation of the communication technology.

II) Universal addressing, since each device is uniquely addressed independently of the used technology. Here, addressing is not defined at technology level, but rather handled by USABle, responsible for mapping each message to the specific technology-level address.

III) Routing between technologies, permitting messages to be delivered across technologies, using the available communication technology and universal addressing.

IV) Pluggable and extensible design, permitting new communication technologies and routing strategies to be added at runtime. Thus, a new routing strategy reuses the existing technologies, without requiring any knowledge of communication-level information.

V) Management interfaces to permit the insertion and control of new technologies and routing algorithms at runtime. Hence, offering to the application mechanisms to inspect the behavior and performance of USABle and adapting it according to changes in the execution environment, developers have control of how USABle behaves at runtime.

In summary, the main contribution of USABle is a modular framework that permits technologies and routing strategies to be plugged and adapted at runtime using management interfaces. USABle offers the following benefits: i) it provides a communication framework that routes messages between different technologies; ii) higher transparency since it abstracts away from developers the complexity of dealing with the communication technologies; iii) it permits routing strategies that use concepts such as multi-hop or carry-and-forward to be more easily implemented and iv) it offers to application developers implementations of five communication

technologies, two well-known multi-hop routing strategies and three carry-and-forward strategies.

Considering the communication technologies available in mobile devices (e.g. NFC, Bluetooth Low Energy, Bluetooth, Wi-Fi Direct, Wi-Fi, UMTS, LTE), they have static information that describes their behavior, such as nominal bandwidth, communication range or security, as well as dynamic information that can only be accessed at runtime, such as latency, packet loss or available devices. Thus, static and dynamic information about each technology are used to guide the development and adaptation of application behavior.

USABle provides implementation that permits communication using Bluetooth, Wi-Fi and Wi-Fi Direct, along with two implementations for mobile devices communicating through cellular networks, namely Google Cloud Messaging (GCM)[4], which is a Google implementation to send push messages from cloud servers to Android-based mobile devices; and Scalable Data Distribution Layer (SDDL)[5] (DAVID *et al.*, 2013), which is a communication middleware to connect fixed and mobile nodes, aiming to permit the diffusion of context data in a scalable manner.

4.2.1.1 Architecture Specification

USABle is based on a modular, extensible and layer-based architecture that permits developers to plug in new communication technologies and routing strategies, reusing much of the existing functionalities. This architecture was inspired by a proposal of (CONTI *et al.*, 2010), following the opportunistic computing paradigm, trying to facilitate the construction of applications that explore physical proximity. Implemented for Android-based applications, using USABle messages exchanged between devices are described using JSON (Java Script Object Notation), a lightweight data exchange format to foster interoperability. Binaries and a series of documents describing the use of USABle are available at the USABle project website[6].

Aiming to improve modularity and adaptability of the framework, USABLe follows the Layered Architectural style presented by (BUSCHMANN *et al.*, 1996) and is divided into three layers: i) Connection layer; ii) Network Layer; and iii) Application-Layer. The two lower layers (Connection and Network) can be adapted or modified at runtime according to application requirements. Figure 13 shows the architecture of the framework and the core design in each layer.

---

[4] https://developers.google.com/cloud-messaging/
[5] http://lac-rio.com/sddl/
[6] http://r2d2.great.ufc.br/dokuwiki

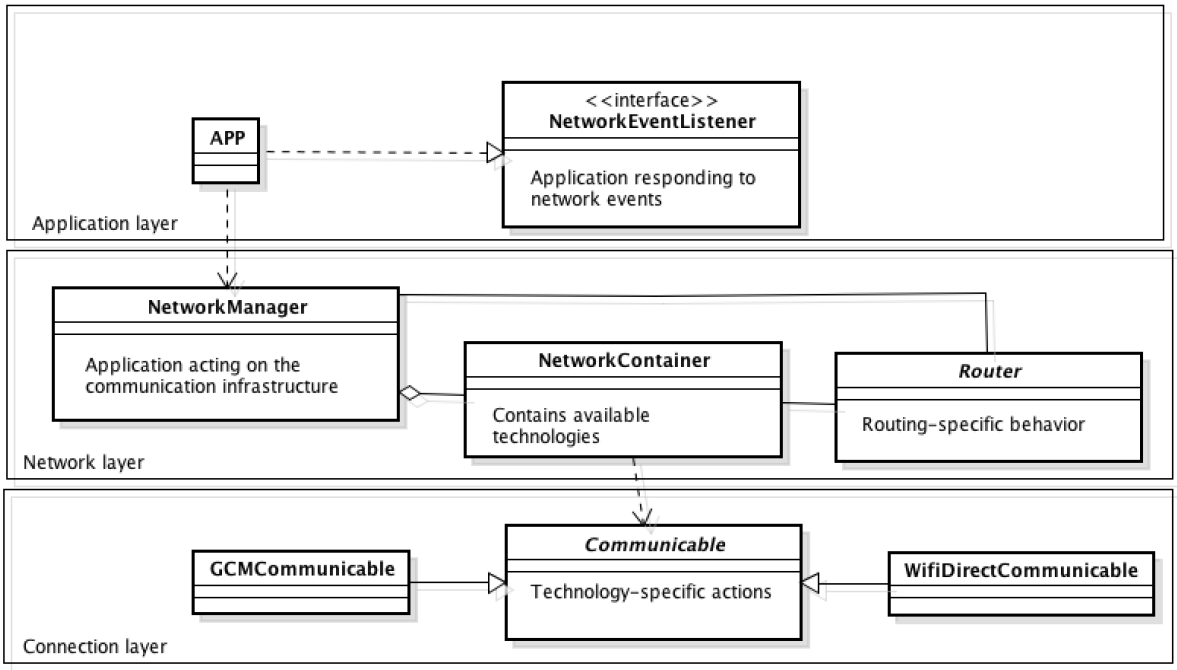

Figure 13 – Architecture of the USABle communication framework.

The connection layer handles the functionalities specific to each communication technology. Each communication module (e.g Bluetooth, GCM) deals with neighbor discovery, connection and disconnection, and message delivery within its own technology. To use a new technology, a *Communicable* abstract class must be implemented and added to the *NetworkContainer* class, in the network layer.

The network layer uses the primitives from the connection layer and implements routing functionalities. Since it executes on top of the connection layer, and has access to all available communication modules, it can route messages between technologies. For instance, one device can receive a message using its Bluetooth interface and send it to a different device using GCM. New routing algorithms must implement the *Router* abstract class.

The concrete class responsible for routing does not have access to the *Communicable* (technologies) classes. Rather, every message is sent using the *NetworkContainer*, which has control of the existing communication technologies. Thus, management actions such as starting and stopping the technologies, as well as replacing the routing algorithm are executed by the *NetworkContainer* class, on behalf of the *NetworkManager* class.

The *NetworkManager* class is responsible for exposing to the application layer two sets of functionalities: i) sending and receiving messages to/from the applications; and ii) management functionalities, to permit the application to add, start, monitor, stop and replace communication technologies and routing strategies.

4.2.1.2 Connection Layer

This layer is responsible for creating technology-specific links between two devices. The main features implemented in this layer are: 1) Neighbor discovery and connection, 2) Message exchange, and 3) Neighbor disconnection. These features are exposed to the upper layers as interfaces and listeners. Each communication technology available is encapsulated as a different module in the connection layer (Figure 13), plugged into the framework and used by the network layer to route and send messages between devices.

Figure 14 shows the interfaces exposed by the connection layer[7]. There are two sets of functionalities: *MessageEventsListener* and *Communicable*. The first set (Figure 4.3, right-hand side), implemented in the *MessageEventsListener* interface, handles events occurring in the communication technology module, such as message received or neighbor found. These events are dealt with using *listeners*, and permit developers of routing strategies and applications to implement specific actions in response to events occurring in the technology module. For instance, once a neighbor connects or disconnects, the neighbor table in the routing layer can be updated. For carry-and-forward strategies, a new neighbor found means that stored messages must be delivered to this new device. Another example is a message arriving at the device, either notifying applications or routing it to nearby devices.

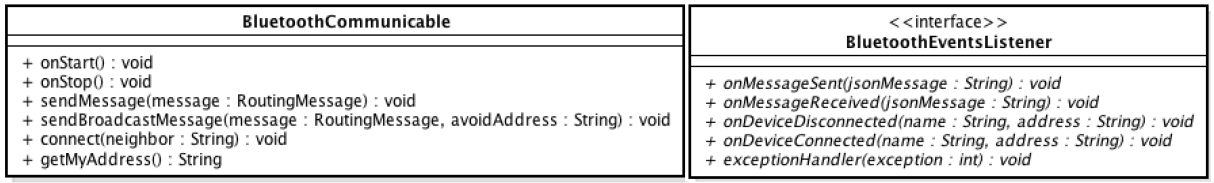

Figure 14 – Interfaces exposed by the Bluetooth communication module.

The second set of functionalities (Figure 4.3, left-hand side), in the *BluetoothCommunicable* class, acts upon each technology module. These functionalities implement management and execution actions such as sending messages, starting and stopping the module. For developers of the uppers layers that use these interfaces, it is sufficient to know how to invoke the interfaces and the effect they produce. For instance, if the routing or application layer needs to send a message to a neighbor device, it would have to invoke the *sendMessage* or the *sendBroadcastMessage* methods, passing the required paramenters. The actual implementation of the module is unknown to the developer.

[7] http://r2d2.great.ufc.br/dokuwiki/doku.php?id=bluetoothnetwork:bluetooth_network

Here, the goal is to hide from developers the complexity to manage each communication technology, since all it is required is the instantiation of the module and the invocation of the appropriate method.

4.2.1.3 Network Layer

Using the interfaces offered by the lower layer, the network layer is responsible for routing messages to their destinations. The goal is to permit communication among devices that are not directly connected, as well as permit the communication using different technologies. Since no supposition about a centralized routing service should be made (SARKAR; LOL, 2010; CONTI *et al.*, 2010), messages are sent using multi-hop or carry-and forward strategies, and all devices running USABle are responsible for forwarding messages.

Figure 15 shows the core interfaces and part of the design of the Network layer[8]. It is worth noting that the routing algorithm takes into account the different communication technologies present in the connection layer. This approach permits messages to be routed from one technology to another, from Bluetooth to WiFi, for example.

The *Router* abstract class is responsible for receiving and forwarding messages, which is the base class for implementing a new routing strategy. This framework offers to developers two implementations of well-known routing algorithms: Flooding (VISWANATH; OBRACZKA, 2006) and AODV (PERKINS; ROYER, 1997).

Upon receiving a message, the *NetworkContainer* delivers it to the *Router* class by the *MessageEventListener* interface, using the *onMessageReceived* method. The *Router* class decides what to do with that incoming message. It may decide that this is a message addressed to an application running on the same device, and the message is given to the *Receiver* class, to be delivered to the applications. Another option is to route the message to another device.

To implement a different routing algorithm, the developer would have to implement the *Router* class. All the functionalities regarding neighbor discovery and selection, and message delivery in the connection layer, along with the interfaces between connection and network layers, are dealt with by the framework and ready to be reused. This approach reduces the complexity of developing a new routing algorithm.

Along with the two implementations multi-hop routing algorithms, three implementations of carry-and forward strategies are available to be reused by developers:

[8] The entire class diagram, among other documents, can be found the project web site

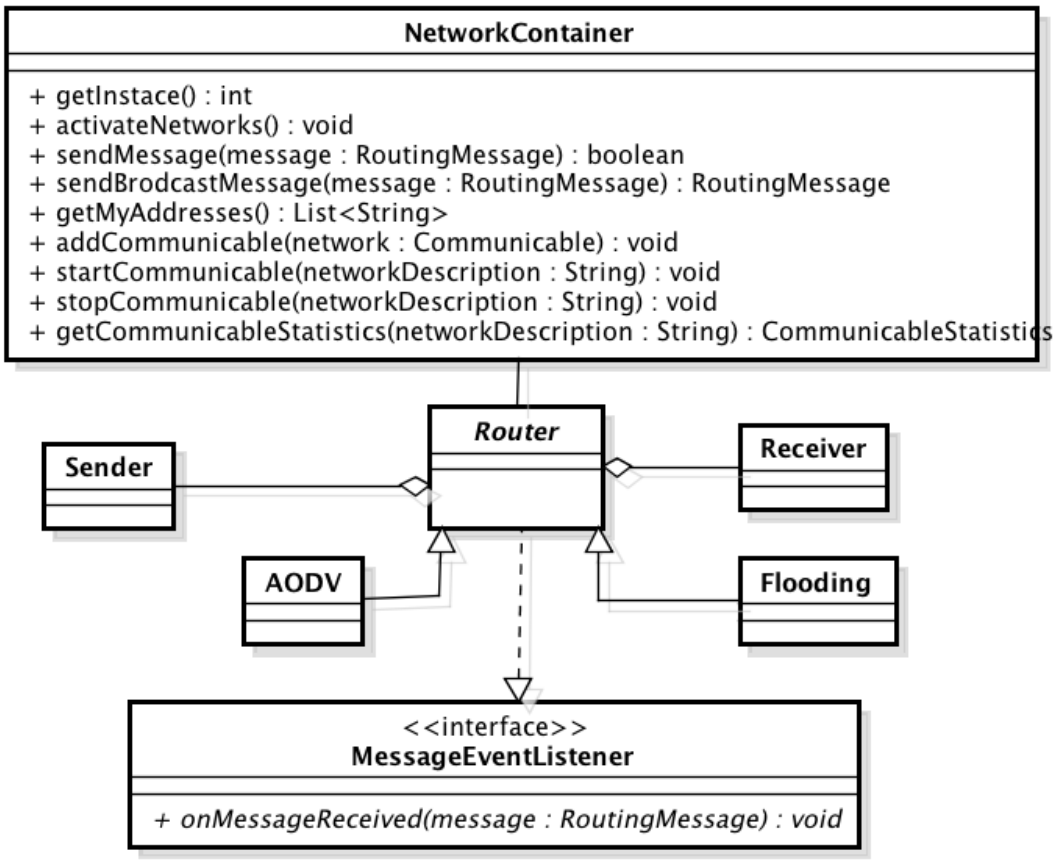

Figure 15 – Interfaces of the network layer

1. Epidemic: Devices carry messages as they move. When two devices meet, they exchange all messages stored in their buffer. *2. Interest-based:* Every message has a tag describing its content. Every device has interests in specific contents. Once two devices meet, they exchange messages based on each others' interest. *3. Interest-based with time-to-live (TTL):* Similar to the interest-based, but every message has a TTL parameter. When a message is retransmitted, its TTL is decremented. Only messages with TTL above zero are retransmitted. This is useful to avoid messages to be retransmitted endlessly.

Figure 16 shows a message stack in USABle. Once a message is received from the technologies modules (GCM and SDDL), the payload is extracted and delivered to the network layer. At this point, the routing algorithm handles the message. It may deliver the payload to the application or wrap it back up down to the available technologies. For each technology, the message from the network layer becomes the payload of the technology module.

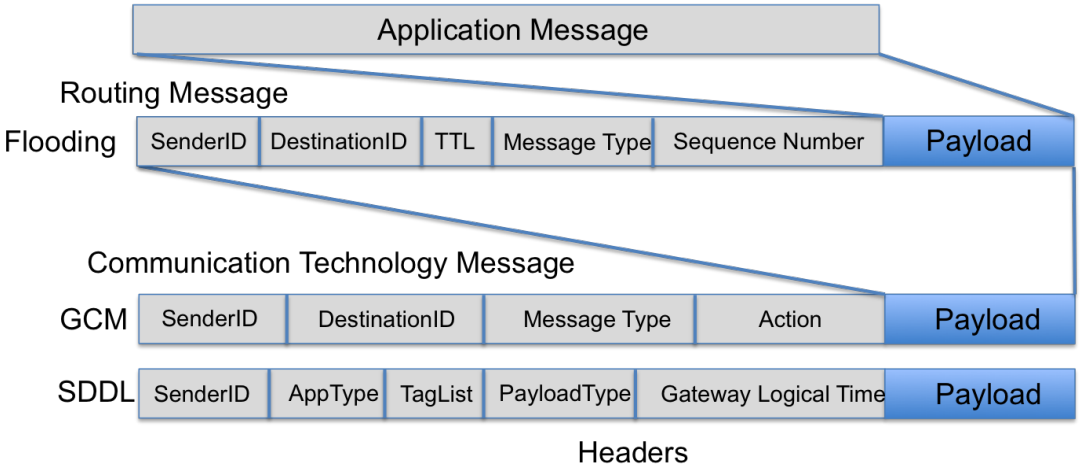

Figure 16 – Message stack in USABle

4.2.1.4 Application Layer

One instance of the communication framework is deployed per device. Abstracting away from the application developer all aspects regarding multi-hop and carry-and-forward on the network layer, and any technology-related functionalities, the framework offers a two set of functionalities: i) To deal with events occurring in the lower layers; and ii) send messages to all reachable devices.

Figure 4.6, in the *NetworkEventListener* interface, presents methods that are used by application developers to implement application-specific behavior in response to events in the network layer, such as new neighbor found or message received[9]. These are callback methods that are executed by the framework when a new neighbor is found or a message is received.

Also in figure 16, the *NetworkManager* class is responsible for sending messages, either unicast of broadcast, and manage the routing algorithm being used.

USABle permits the management and adaptation of the connection layer and of the network layer. Management of the connection layer is comprised of adding, starting and stopping technologies modules.

Another management functionality is the *getCommunicableStatistics*, from the *NetworkContainer* class (Figure 4.6). This method returns statistics about message loss and delivery time. Periodically, the network layer sends *Hello* messages to its neighbors using the available communication modules. The network layer keeps track of each module's performance (message loss and delivery time). This information permits applications to decide which communication modules they want to use at a given moment.

[9] http://r2d2.great.ufc.br/dokuwiki

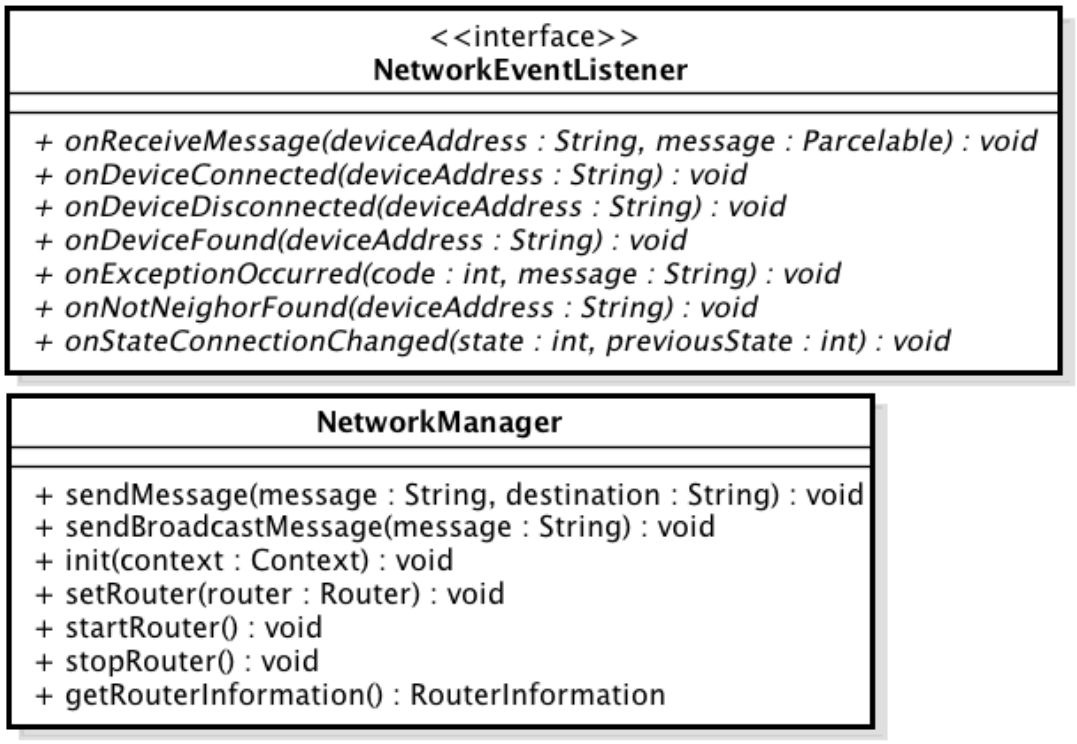

Figure 17 – Interfaces exposed by the network layer to the application layer

Management of the network layer is responsible for deciding at runtime the routing algorithm to be used. Class *NetworkManager* has methods to add, start and stop the routing algorithm in the local device (this action has no effect on neighbouring devices). Each routing algorithm must implement an *IRouter* interface containing methods used by the *NetworkManager* class. Since there are several routing algorithms available, the actual implementation of the *IRouter* interface made by each routing algorithm hides specific actions for each algorithm (e.g. initialization, execution, finalization).

4.2.2 Decentralized Coordination Infrastructure

USABle permits devices to interact based on message exchange. However, direct interaction is not suited in highly dynamic environments, since devices may enter and leave the network unpredictably, and assumptions on the availability of resources are difficult to be guaranteed. On the other hand, to handle the dynamism of CPS, the coordination model based on shared data space is better suited (MAMEI; ZAMBONELLI, 2009a). Shared data space is an independent associative memory and shared among all nodes of the system. Communication between processes takes place via the insertion and reading of tuples in this shared space.

On top of the communication infrastructure, this doctoral thesis uses a distributed co-

ordination mechanism to minimize the coupling among interacting entities. To offer coordination mechanisms in CPS, an event-based model is also implemented using the shared data space (tuple space). Using tuples and events, coordination is accomplished using an infrastructure called System Support for Ubiquity - Distributed Tuple Space (SysSU-DTS) (NETO *et al.*, 2013b).

The main entities of SysSU-DTS are App, UbiBroker and UbiCentre. An App represents applications or services coordinating their activities. UbiCentre is an entity representing the tuple space, and allows tuples to be concurrently accessed by Apps through an UbiBroker. UbiBroker is responsible for giving access to tuple space operations.

These operations are divided in two sets: i) act upon the tuple space, comprised of the put (place tuples in the tuple space), read and take (read and remove the tuple) operations; and ii) pub-sub operations to subscribe and unsubscribe to events on the tuple space. These operations are explained more detailedly in section 4.3.3.

Tuple-spaces may be hosted in two different devices: i) fixed server and ii) mobile device. Tuple-spaces running on a fixed server are accessed directly using the address of the server. Tuple-spaces running on mobile devices are accessed using the USABle communication framework. Figure 4.7 shows several tuple spaces interacting using USABle.

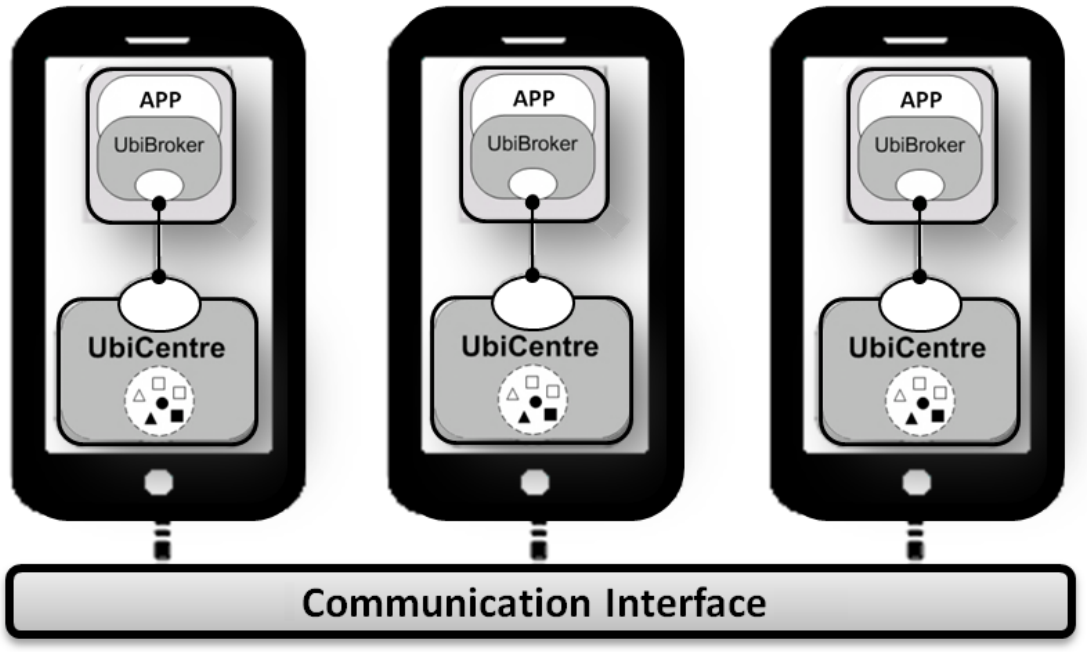

Figure 18 – Several distributed tuple spaces interacting using USABle.

Using the communication interface, a component called TS-Monitor is implemented, responsible for self-adapting the tuple-space and communication infrastructure. TS-monitor has a list of tuple descriptors, which are informed by local applications the list of tuples they

are interested in. It also proactively monitors the available devices, looking for tuples with an specific description (CIB), matching the descriptors informed by the applications. Whenever two devices come into contact, they exchange the list of tuples descriptors they provide. Thus, one tuple space knows where to find remote tuples with specific descriptions.

Additionally, put, read and take operations generate effects on neighbors within an specific TTL value. Whenever a tuple is inserted in a tuple space, its descriptor is propagated to neighbors. Read and take operations return tuples present in neighbors as well.

Shared spaces are difficult to implement in highly mobile/volatile environments, mainly because of the complexity to maintain a consistent view across all shared spaces (CURINO *et al.*, 2005). Here, this problem is minimized because of two reasons: i) a reactive approach is used (DE *et al.*, 2013), meaning that when a put operation is executed in a device, that tuple is diffused to interested nearby devices. Additionally, read operations are propagated to nearby devices, where only tuples written in that device within a time frame are considered valid. ii) each device has an individual tuple space. There is no assumption on tuples spaces being consistent.

Scope is a concept used to restrict the communication between devices, to forward tuple descriptors and content only to interested devices. For instance, a tuple scope can be limited to co-located devices in a network range (physical location); similarly, scope can limit tuple visibility to users or applications with specific characteristics, such as only professors or only students (logical location). The idea is to create the concept of co-location (physical or logical), where devices are sharing a situation and only forward tuples within a given scope.

4.3 Execution and Adaptation

Execution of self-adaptive CPS is composed of the following requirements; i) communication and coordination, presented in the previous section; ii) sensor and actuator modeling and access; iii) context modeling and access; iv) and application management and adaptation. Among these requirements, adaptation plays a central role, permitting to create systems capable of adapting their behavior and structure to deal with changes in the execution environment.

CyberSupport uses components to encapsulate access to and control of sensors and actuators, to provide meaningful context information obtained from sensor data and inferred using context management infrastructures. Finally, adaptation takes place by changing components at runtime.

CPS adaptation based on components is based on three requirements: i) existence of a sensor and actuator description mechanism; ii) existence of context information describing system execution; and iii) possibility to change component instance at runtime.

4.3.1 Sensor and Actuator Description Model

The description of sensors and actuators is important to discover and access resources available in the environment at runtime. That description should consider two characteristics of CPS: i) they are formed by several distributed subsystems, making interoperability an important requirement; ii) the number and device capabilities vary unpredictably, requiring the description formalism to be as light as possible.

With interoperability and lightness as design principles, a vocabulary to represent sensor and actuator resources is used both at design-time to describe required and provided functionality, and at runtime to describe and discover available resources.

The vocabulary available to represent contextual information is organized as an information tree. It can be referenced by a sequence of node names separated by dots. A similar mechanism is found in the network management domain, the Management Information Base (MIB)[10]. Figure 4.8 shows the hierarchical vocabulary called CIB (Contextual Information Base). For instance, the temperature in a given ambient can be accessed as 'context.ambient.temperature'. This approach helps to uniquely identify the type of sensor and actuator information being used. This base must be kept in a centralized repository where developers have access and can maintain the list of existing sensors and actuators.

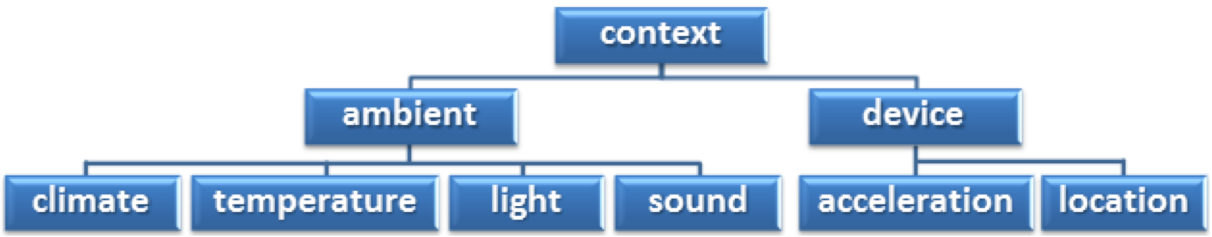

Figure 19 – Subset of the information tree in the CIB.

The following definitions specify the concepts used to describe sensors and actuators.

Definition 1. Key. A key is a path from the root to a specific leaf in the CIB tree, uniquely identifying a resource in CPS. Thus, the CIB links several subsystems interacting by message exchange. The primary concern is to create a light formalism to support the description

[10] http://www.ieee802.org/1/pages/MIBS.html

of sensors, actuators and the inference of more complex context situations. The Key definition is formed by the following rule:

concept ::= (Set of CIB concepts)

Key ::= <concept> (‘ ’ | ‘.’ <Key>)

Definition 2. Type. It describes the data type exchanged between systems, sensors and actuators. Types are only of primitive types, as follows (LIMA *et al.*, 2011a):

1. string: a sequence of characters enclosed by quotation marks;
2. integer: an arbitrary integer number;
3. float: a floating-point arbitrary number;
4. boolean: a logic number represented by true of false;
5. array: an ordered sequence of similar primitive types separated by a comma;

Every Key concept has an associated Type. The relation between Key and Type is described in the CIB tree. For lightness, at runtime the Type of a CIB is omitted. That information is present in the CIB tree and agreed upon by every developer when using a CIB.

Definition 3. Value. Value is the raw data exchanged between systems, sensors and actuators. It always has a specific type.

Definition 4. Data. Data is every piece of information read from a sensor or fed to an actuator. Data is described as *(key, value)* pair, according to the following rule:

Data::= (<key> ‘, ’ <value>)

Definition 5. Tuple. Tuple is a sequence of a single or more Data. Complex types are created using several Data types. For instance, a profile Tuple may be formed by the following Data sequence: (context.user.name, "Marcio Maia"), (context.user.location.latitude, 3.3456533), (context.user.location.longitude, 4.3456533). The rule describing a Tuple is as follows:

Tuple := <data> (‘ ’ | ‘, ’ <Tuple>)

Definition 6. Label. Label is defined by a Universal User Id (UUID)[11], which is a formalism to describe a sequence of numbers and characters that uniquely identifies a resource.

Definition 7. Resource. The execution environment is formed by resources (sensors and actuators) and software services. For simplicity, a resource is fully described by a Key and a Label, as follows:

Resource::= (<Key> ‘,’ <Label>)

Definition 8. Resource properties. Each resource, service or component may have

[11] http://www.opengroup.org/dce/

execution characteristics described using a resource descriptor file. The type and format of the characteristics is application-dependent. For instance, a position sensor may have an accuracy parameter that helps applications to choose which sensor to use at a given moment. Additionally, a response time of a remote service may turn the service not tailored to a given activity.

Definition 9. State. State is a formalism to describe a relevant situation to CPS. It may be simple information directly acquired from sensors, such as a GPS location, or may be complex situations inferred from data received by several sensors. At runtime, the State is represented by a Tuple.

Definition 10. Software component. A component is a piece of software that executes actions on a given environment. There are two types of components: i) sensing components, responsible for reading information about the environment, are described by a Resource, an output parameter (Tuple, mandatory) and input parameter (Tuple, optional); and ii) actuating components, responsible for acting upon the environment, are described by a Resource, an output parameter (Tuple, optional) and input parameter (Tuple, mandatory). A component may have a property descriptor as well, to permit the description of its non-functional characteristics requirements.

4.3.2 Component Model

A Context-Actuator Component (CAC) is responsible for accessing sensors, actuators, other CACs and services. This component model extends the component model presented in (MAIA *et al.*, 2013) and aims to foster reuse and facilitate the addition and replacement of new components. The goal is two-fold: 1) to permit the life-cycle management of each CAC based on the component model presented in Gui et al. (GUI *et al.*, 2011) and 2) to force interaction using the tuple-space to minimize coupling.

Figure 20 presents the main interfaces in a CAC. The *start and stop* interfaces are responsible for initiating and terminating the component execution. The *isRunning* interface verifies if the component is executing. The *setProperty* interface receives as input the Data parameter, to access a sensor or control an actuator. The *getPropertiesKeys* returns all keys defining the configurable properties of a CAC. The *publish* interface required by each CAC is responsible for publishing information in the tuple-space. A CAC must access the UbiBroker interface of SysSU-DTS in order to publish or query information on the tuple space.

Each CAC has a manifest file, which contains identifiers of the information required

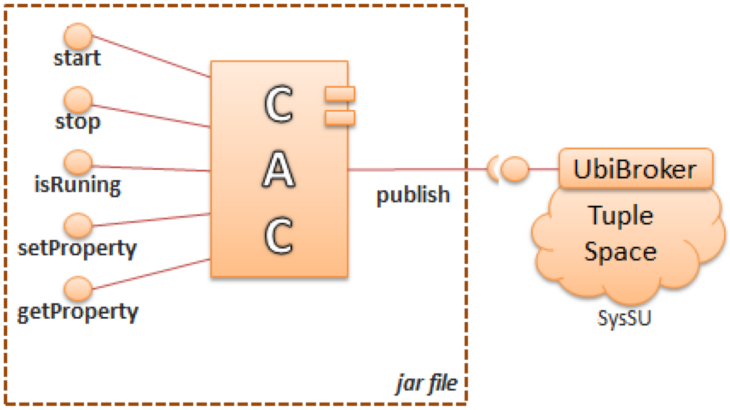

Figure 20 – CAC Model (FONTELES *et al.*, 2013).

or generated by the CAC (i.e., keys from the CIB vocabulary).

4.3.3 Development and Execution model

To improve the coupling among system modules, the development and execution model in CyberSupport was based on tuple spaces and Publish/Subscribe primitives, carried out by SysSU-DTS. In that direction, according to the component definition (Definition 10), any interaction between sensors, actuators, components and applications is carried out based on tuples.

4.3.3.1 Matchmaking and Reactions

SysSU-DTS offers functionalities to act upon the tuples present in tuple spaces, such as tuple matchmaking and reaction mechanisms. When data from sensors is generated, it is placed in SysSU-DTS as a tuple with certain characteristics. Once a tuple with a given characteristic (key) is inserted, interested components are notified about the existence of this tuple.

The matchmaking mechanism is used when reading and removing tuples, created to permit developers to specify the characteristics of the required tuples. It returns a subset of tuples satisfying a given query. A *Query* is composed of a tuple pattern and a filter. A pattern describes the format, value or data type present in a tuple. For instance, a tuple with the pattern {(null, string),(null, string)}, represents tuples with two string fields, whereas {(contextData, "contex.ambient.relative-humidity"), (value, null)}, comprises tuples with any value of relative humidity. Algorithm 4.1 shows a pattern used for matching any values in the tuple.

Algorithm 1 – Tuple Pattern

```
1 //Tuples matching any of the fields
2 Pattern pattern = (Pattern) new Pattern()
3 .addField( "ContextKey", sensorDescription )
4 .addField( "Source", "?" )
5 .addField( "Values", "?" )
6 .addField( "Accuracy", "?" )
7 .addField( "Timestamp", "?" );
```

Additionally, a filter is a boolean function, in which only tuples satisfying this function are selected. Filters are written using Java classes, which improves expressivity to specify relations between tuple fields. This happens because the developer can implement the matchmaking, and matching between tuples can be specified using a programming language.

For instance, the matchmaking mechanism permits to select tuples with contextual information regarding temperature, collected by the device thermometer, reading a temperature value above 25 degrees celsius. To do so, it is necessary to create a query composed by the pattern {(contextData, "contex.ambient.temperature"), (value, null)}, and by the filter "function filter(tuple) {return (tuple.value > 25 && tuple.source == 'internalThermometer')}". Algorithm 4.2 shows a filter implemented in Java.

Algorithm 2 – Tuple filter

```
1  //Tuples with temperature value higher than 10
2  IFilter filter = new IFilter.Stub() {
3      public boolean filter(Tuple tuple) throws RemoteException {
4
5          for(int i = 0; i < tuple.size(); i++) {
6
7              if(tuple.getField(i).getName().equalsIgnoreCase("Values")) {
8                  List values = (List)tuple.getField(i).getValue();
9                  float temperature = Float.parseFloat( (String)values.get(0) );
10                 if( temperature > 10 ) return true;
11
12             }
13
14         }
15
16         return false;
17     }
18
19 };
```

The event mechanism has a feature called reaction, which is executed when a given event is detected. An event is detected using the matchmaking mechanism. Applications may subscribe to receive notification of events, informing a Query (matchmaking, Algorithm 4.2) and a Reaction. This Reaction is executed whenever a tuple satisfying the informed Query is inserted in the tuple space. Algorithm 4.3 shows a reaction implemented in Java.

Algorithm 3 – Tuple Reaction

```
IClientReaction reaction = new IClientReaction.Stub() {
    public void react(Tuple tuple) throws RemoteException {
        //Application-specific Actions to be executed when a tuple is found matching the pattern and or filter.

    }
};
```

The matching and reactions mechanisms are used to implement actions executed when a system enter or leave a specific state. This approach permits developers to implement actions in response to events in the execution environment.

4.3.3.2 SysSU-DTS operations

As mentioned in section 4.2.2, SysSU-DTS is comprised of two sets of operations: i) act upon the tuple space, comprised of the put (place tuples in the tuple space), read and take (read and remove the tuple) operations; and ii) pub-sub operations to subscribe and unsubscribe to events on the tuple space (Algorithm 4.4, readSync and takeSync). Algorithm 4 shows the interfaces exposed by SysSU-DTS to the applications.

Algorithm 4 – SysSU-DTS interfaces

```
public interface SysSUDTS{

    //Write a tuple in the tuple space
    public void put(Tuple tuple);

    //Read synchronously and assynchronously
    public List<Tuple> read(Pattern pattern, IFilter filter);
    public List<Tuple> readSync( Pattern pattern, IFilter filter, long timeout);

    //Remove synchronously and assynchronously
```

```
11      public List<Tuple> take(Pattern pattern, IFilter filter);
12      public List<Tuple> takeSync( String key, IFilter filter, long timeout );
13
14      //Subscribe and unsubscribe to events in the tuple space
15      public String subscribe(IClientReaction reaction, String operation,
16                                                  String key, IFilter filter);
17      public void unSubscribe(String operation, String key);
18
19  }
```

The read and take operations access the tuple space to read tuples described by the pattern and filter classes, and a timeout value is informed to be used by synchronous operations. The difference between those two is that the take operation removes the tuple from the tuple repository.

The subscribe operation follows the pub-sub paradigm to implement specific functionality specified by the reaction object in response to an given operation (read, take, put), along with the CIB key and filter classes.

4.3.3.3 Interaction

The interaction model is based on information placed on the tuple space. Every action requiring input and output parameters takes place using tuples and SysSU-DTS. For instance, reading the light level from a sensor is executed reading a tuple in SysSU-DTS with the pattern {(contextData, "context.ambient.light") }. A CAC (Figure 20, setProperty method) running on the device or on other devices capable of providing light level respond by placing the corresponding value in the tuple space ({(contextData, "context.ambient.light"), (value, 1)}). Using this approach, there is no direct invocation from application to sensor. Everything takes place solely using the CIB concept (context.ambient.light).

The interaction between application and sensors/actuators is based on get/set methods using the tuple space. Information from sensors is read using get methods, specifying the information description ("context.ambient.light"). Similarly, controllers are accessed informing the specific description and passing the value to be set.

Algorithm 5 – Invoking sensors and actuators.

```
1  public get/setOperation(){
2
3      String sensorDescription = "context.user.id";
```

```
4       String controllerDescription = "controller.ambient.temperature";
5       String controllerDescriptionResult = "controller.ambient.temperature.↩
            result";
6
7       long timeout = 2000;
8
9       //Setting a temperature to 20 degress
10      ArrayList<String> getValues = new ArrayList<String>();
11      getValues.add( "idMarcioMaia" );
12
13      Tuple getInput = (Tuple) new Tuple()
14      .addField("AppId", "Cybersupport" )
15      .addField("ContextKey", sensorDescription )
16      .addField( "Values", getValues );
17
18      //Synchronous set
19      //filter is a parameter discribed in Algorithm 3.2
20      List tupleList = loCCAM.synchronousGet( getInput, filter, timeout );
21
22      //Setting a temperature to 20 degress
23      ArrayList<String> temperature = new ArrayList<String>();
24      values.add( "20" );
25
26      Tuple setInput = (Tuple) new Tuple()
27      .addField("AppId", "Cybersupport" )
28      .addField("ContextKey", controllerDescription )
29      .addField( "Values", temperature );
30
31      //Synchronous set
32      List tupleList = loCCAM.synchronousSet( setInput,
33                                          controllerDescriptionResult, timeout );
34
35      //Asynchronous get/set
36      //reaction is a parameter discribed in Algorithm 3.3
37      loCCAM.asynchronousGet( inputGet, reaction, filter );
38      loCCAM.asynchronousSet( inputSet, reaction, controllerDescriptionResult );
39
40  }
```

Algorithm 5 shows an example of the steps necessary to invoke sensors and actuators, based primarily on the description of the required sensor or actuators. Sensors and actuators can be accessed synchronously or asynchronously. The main difference between those two is that while the synchronous invocation is based on a timeout value, restricting the time get/set operations require, the asynchronous invocation does not have an execution upper bound. Instead,

it requires a reaction, which is a mechanism to let developers specify application-specific behavior to respond to events (Algorithm 4.3).

Synchronous get invocation requires a filter to match tuples according to specific characteristics, a sensor description and a timeout value. Synchronous set requires a tuple write the value in, the description of the result information, in case a component needs to return a value, like a result code or some exception description, and a timeout value. The result of synchronous invocation is the list of tuples found in the get methods and the return in set methods. Additionally, as previously mentioned, asynchronous get/set requires a reaction parameter instead of a timeout value. They do not return any value, rather the found tuples are accessed in the *react* method, in the *Reaction* class.

Algorithm 6 – Synchronous get/set.

```
1  public List<Tuple> synchronousGet( Tuple setInput, IFilter filter,
2                          long timeout ) throws TupleSpaceException {
3
4       //Getting the value from the CAC described by the key
5       List<Tuple> tupleList = sysSUStart.readSync( setInput, filter, timeout );
6       return tupleList;
7  }
8
9  public List<Tuple> synchronousSet( Tuple inputTuple, IFilter filter,
10      String returnKey, long timeout  ) throws TupleSpaceException {
11
12    //Setting the input tuple in the Tuple Space. The CAC will
13    //respond and set the controller
14    sysSUStart.put( inputTuple );
15
16    //Reading the results from the set operation, placed in the TS
17    //by the CAC, whenever the result is ready
18    List<Tuple> resultList = sysSUStart.readSync( returnKey, filter, timeout );
19
20    return resultList;
21
22 }
```

Algorithm 6 shows the methods to implement the synchronous get and set methods. The developer should provide a filter (to select specific tuples) and the CIB key describing from which sensor the data is to be read. On the other hand, the synchronous set method requires the input tuple, along with the filter and CIB key for the result of the set method invocation. After

the tuple is put in the tuple space, it will be read by the actuator defined by the CIB key present in the tuple, followed by the actuator execution. The result is placed in the tuple space by the actuator component using the return key.

Figure 21 shows the message flow when an application invokes a component using synchronous get/set. The first action is the execution of the *synchronousGet/Set*, by placing a tuple in SysSU-DTS, and the consequent blocking of the application flow. When that tuple is placed in SysSU-DTS, one of two possible actions is executed. The first is when a CAC is found in the component management space. Thus, that component is invoked, which means that environment resource will be accessed and controlled. Although the application invoked the component using a synchronous invocation, the communication between CAC and the actual resource takes place asynchronously. That approach facilitates the infrastructure to deal with disconnections and runtime replacement.

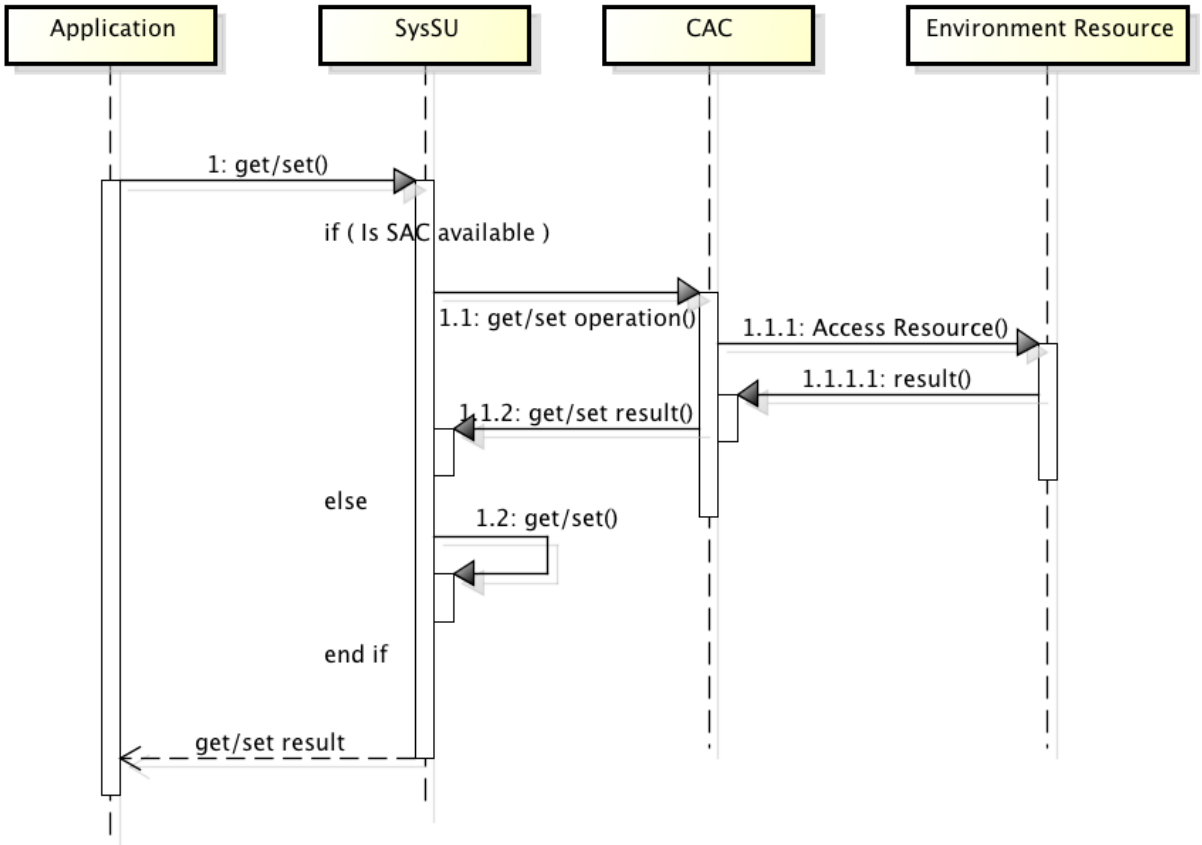

Figure 21 – Message flow using synchronous get/set invocation

The second path is when the correct CAC is not found. The same parameters to invoke the CAC locally are passed to the communication and coordination infrastructures to look for that CAC in nearby devices. Here, since the information from sensors and to actuators

is collected from distributed devices, issues regarding the consistency and coherence of tuples must be considered. The solution is to enforce that distributed queries always generate an access to a sensor or actuator (no cached value is returned). Additionally, to guarantee the quality of the read/controlled values, accuracy descriptors (expiration time, precision) are passed along with the distributed query and only the information fulfilling the accuracy test is returned. To conclude the execution flow, the operation result is pushed back to the application or a timeout exception is thrown.

Opponsingly, asynchronous invocations offer a more uncoupled interaction between applications, sensor and actuator management infrastructure and the actual hardware responsible for sensing or controlling the environment. First, asynchronous get passes the CIB key describing the component to be accessed (sensor), a filter to select specific values only (event description) and a reaction, to execute a given functionality when the information is ready to be delivered. Second, asynchronous set is executed using two separate actions: 1) pass the input tuple to the specific controller (the tuple has the input values and the CIB key describing the controller); 2) subscribe to be notified when the action is completed, using the CIB key describing the result, a filter to match those tuples with specific characteristics (result code, result values, exceptions) and a reaction, to implement application specific actions in response to events in operation execution. Algorithm 7 shows the implementation of the get and set asynchronous invocations.

Algorithm 7 – Asynchronous get/set.

```
1  public void asynchronousGet( Tuple inputTuple, String resultKey,
2                               IClientReaction reaction, IFilter ifilter ) {
3     //Setting the input tuple in the Tuple Space. The CAC will
4     //respond with the get information, based on the input information
5     if( inputTuple != null ) sysSUStart.put( inputTuple );
6
7      //Getting the value from the sensor as a result of the in
8      subscribeLoCCAM(reaction, resultKey, "put", ifilter);
9  }
10
11 public void asynchronousSet( Tuple inputTuple, String resultKey,
12                               IClientReaction reaction, IFilter ifilter ) {
13
14    //Setting the input tuple in the Tuple Space. The CAC will
15    //respond and set the controller
16    sysSUStart.put( inputTuple );
17
18     //Reading the results from the set operation, placed in the TS
```

```
19      //by the SAC, whenever the result is ready
20     subscribeLoCCAM( reaction, resultKey, "put", ifilter );
21
22  }
```

Figure 22 shows the message flow when an application invokes a component using asynchronous get/set. When invoking a component, the application passes a reaction, which will be executed when the action is completed. The react method (within the reaction) receives a tuple as parameter, which has information about the execution of the get/set invocations. Other than the replacement of the timeout value with the reaction parameter, the rest of the execution is somehow similar.

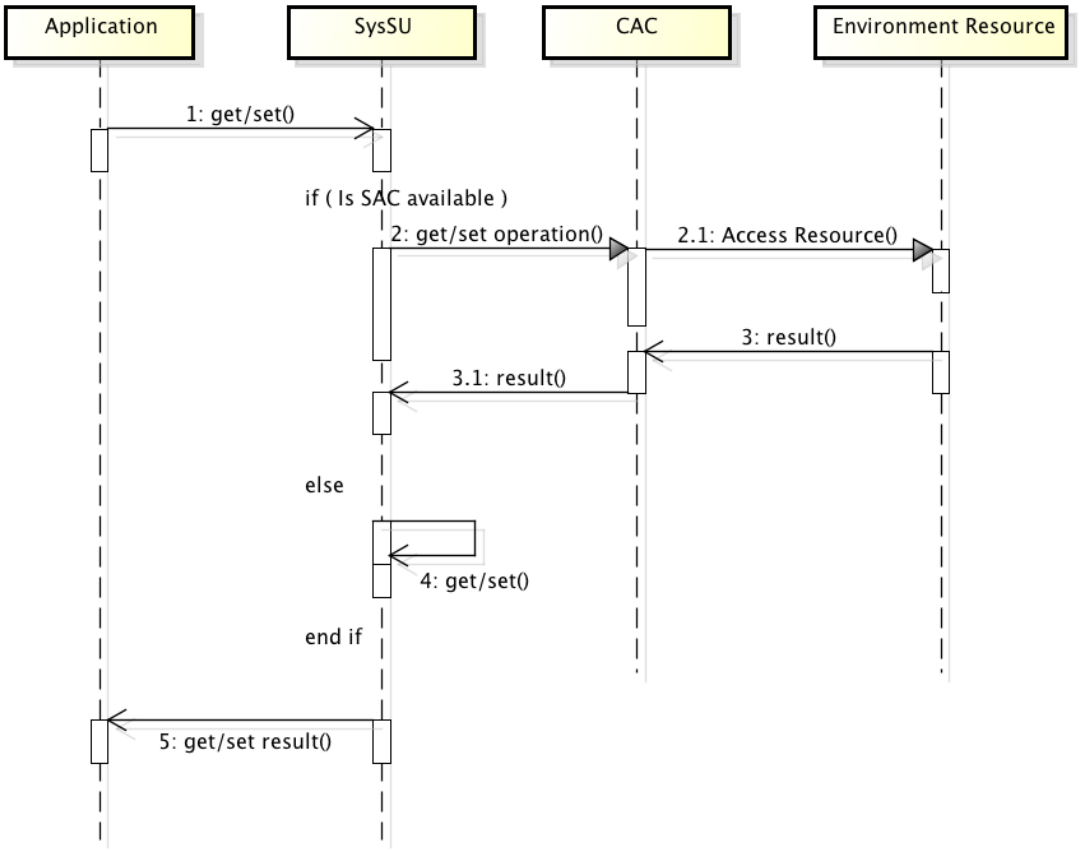

Figure 22 – Message flow using asynchronous get/set invocation

The use of the tuple space to permit the interaction between application and sensor/actuators is mainly to promote uncoupling. In that direction, application development is based on three characteristics: i) common description of the sensors and actuators, ii) known get/set interfaces and iii) agreed data format. These characteristics improve flexibility and robustness, since changing and replacing sensors/actuators require only replacing the actual component

implementation, without having to change the application itself.

Message exchange between distributed devices takes place using the tuple space as well. Direct messages (unicast) are sent using the address of the destination as a tuple ({(contextData, "context.device.directMessage"),(value, destinationId), {(contextData, "context.device.messageContent"),(value, "Message Content")}. A similar approach can be used to send broadcast messages. Similarly, other parameters can be used, as to use a tag or a topic to describe a message. Finally, there are situations where synchronous message exchange is required and SysSU-DTS has read and write blocking operations. Thus, a message is sent and a confirmation must be received before the operation is finished.

Although this separation between interacting entities is useful to improve uncoupling, developers may choose to use the communication framework to send messages directly to other devices. It depends basically on the uncoupling level required by the application. Every message exchanged without shared-spaces or event-based approaches increases the coupling among interacting entities.

4.4 GREat Tour App Implemented using Cybersupport

Cybersupport has been used to reimplement the GREat Tour app, allowing it to become self-adaptive. Any functionality accessing sensors and actuators is implemented encapsulated as CACs. Thus, to change an implementation of a given functionality (e.g. indoor location, from based on QR-Code to NFC Tags), all it takes is to unsubscribe from one CAC and subscribe to another. Figure 23 shows the available CACs being used in the Cybersupport implementation of GREat Tour.

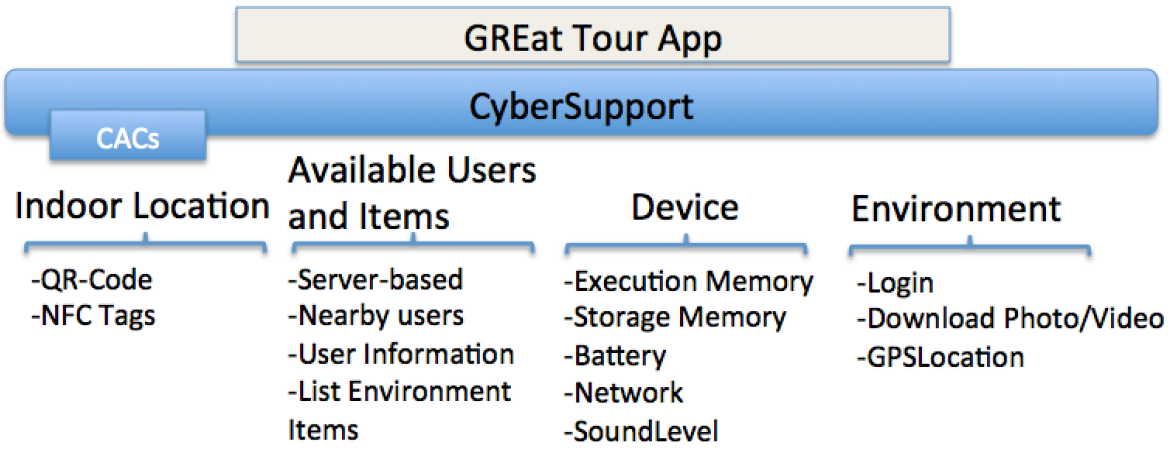

Figure 23 – Great Tour App, Cybersupport and available CACs

The development of self-adaptive CPS applications using Cybersupport is comprised of the following activities, shown in Algorithm 8: i) implementing a reaction, to execute a given functionality when a specific event occurs (lines 5 - 28); ii) specifying the context key, which is the event the reaction is responding to (lines 33, 34); iii) specifying input parameters (lines 37 - 47) and iv) invoking the corresponding get/set methods (lines 49 - 50).

Algorithm 8 – GREat Tour App acessing available users

```
1  public void getAvailableUsers( String roomId ) {
2
3      final static ArrayList users = new ArrayList<Person>();
4
5      IReaction reaction = new IClientReaction.Stub() {
6
7          //The CAC writes to the tuple space the user list in a given ↩
               environment
8          //The react method is executed when the tuple written to the TS ↩
               matches the
9          //context key passed in the get methods
10         public void react(Tuple arg0) throws RemoteException {
11
12             //Getting the list from the retrieved tuple
13             List<Object> userList = (List<Object>)arg0.getField(2).getValue();
14
15             synchronized (users) {
16
17                 for (int i = 0; i< userList.size();i++){
18                     Person person = new Person();
19                     //Adding information from the user to the Person object
20                     //. . .
21                     //users.add(person);
22
23                 }
24
25             }
26         }
27
28     };
29
30     ArrayList<String> getValues = new ArrayList<String>();
31     getValues.add( "CPS Research Room" );
32
33     String serverAvailableUsers = "context.environment.availableUsers.server"
34     String nearbyAvailableUsers = "context.environment.availableUsers.nearby"
35
```

```
36      //The context key describing the user list based on a server
37      Tuple serverGetInput = (Tuple) new Tuple()
38      .addField("AppId", "Cybersupport" )
39      .addField("ContextKey", serverAvailableUsers )
40      .addField( "Values", getValues );
41
42      //The context key describing the user list based on nearby users using
43      //SysSU-DTS and USABle
44       Tuple nearbyGetInput = (Tuple) new Tuple()
45      .addField("AppId", "Cybersupport" )
46      .addField("ContextKey", nearbyAvailableUsers )
47      .addField( "Values", getValues );
48
49      loCCAM.asynchronousGet( serverGetInput, serverAvailableUsers, reaction, ↩
            null );
50      loCCAM.asynchronousGet( nearbyGetInput, nearbyAvailableUsers, reaction, ↩
            null );
51
52  }
```

All interactions between applications and sensor/actuators components are carried out using the tuple space. Here, according to Algorithm 8, the relation between application and sensors/actuators is based only on the description of the required sensors (lines 33 - 34) and the input parameters (lines 37 - 47). Appendix A has a discussion about modularization and the importance of minimizing coupling between application modules.

This approach facilitates the adaptation of the application regarding sensor/actuators access, since changing from one CAC to another requires only to change the context key (lines 33 and 34). The LoCCAM middleware is responsible for managing the component life-cycle (e.g. changing from indoor location based on QR-Codes to based on NFC Tags).

The CAC to access the user list in a room is implemented using two different approaches. First, that list is retrieved using a web service using the room name as input. Similarly, the second approach also uses the room name as input, but uses the USABle communication framework presented in subsection 3.2.1 is used. A query requesting the available users is broadcast. Any device within 3 hops (configuration parameter that can be changed) responds to that query.

Figure 24 summarizes how the nearby CAC collects information about nearby users. Initially, it sends a message requesting information about nearby users (1). That message can be sent using two different mechanisms: i) broadcast to nearby devices, when the CAC writes in

the tuple space a tuple with the nearby users CIB (Algorithm 4.7, line 34); or ii) when a nearby device is detected and the nearby user CIB has already been writen previously.

At this point, a request is sent to the network using USABle. CACs running on nearby devices respond (3) to requests when a nearby user CIB is written on their local tuple space using the "put" operation (2). The device who initiated the request receives the information about nearby devices (4). Now, the application requesting the information is notified using the reaction mechanim (Algorithm 4,7, line 50).

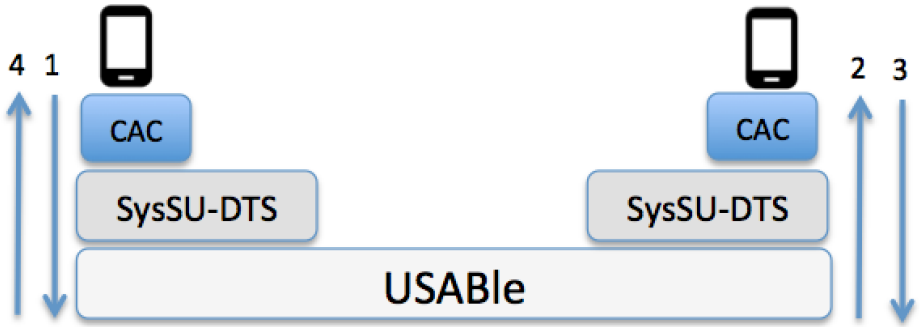

Figure 24 – Highlighting the use of the nearby CAC

Message routing between individual SysSU-DTS is defined by the routing algorithm being used by USABle. The nearby CAC implementation uses the Flooding algorithm, which means that devices outside the current room the user is in, but within communication range, may receive the request as well. The request carries the room description, and only the devices in the same room respond to the request. The room description uses the Scope definition in section 4.2.2.

4.5 Conclusions

This chapter presented an infrastructure to aid developers implement decentralized uncoupled self-adaptive CPS. Decentralization and self-adaptation are driving forces to create the CyberSupport infrastructure. In that direction, it provides mechanisms to let legacy and novel systems, formed by distributed devices and subsystems, to interact based on a light and common formalism.

To handle the different requirements and challenges of self-adaptive CPS, the software stack is divided in two layers: i) Communication and coordination; and ii) Execution and adaptation.

The communication module offers mechanisms to let developers exchange and route messages between distributed devices, being transparent to communication technologies and routing activities, and providing management interfaces, to adapt the entire communication infrastructure.

The coordination module is constructed based on decentralized tuple spaces. Reading and writing operations are executed first on local tuple spaces, but also use tuple spaces present on nearby devices and on servers present on the Internet. On top of the tuple space, a pub/sub mechanism offers filter and reaction primitives to implement response to environment events.

Using the underlying communication and coordination layer, the Execution and adaptation layer permits the specification of application behavior based on a set of actions. Each action is specified using a component model and executed using components present on decentralized devices.

According to the MAPE-K loop, the Monitor and Execute phases are implemented by accessing sensors/actuators, and by notifying the Analyze phase when a given state is entered. To change the sequence of actions to be executed requires only a different Reaction implementation or changing CAC being used.

5 PERFORMANCE EVALUATION

This chapter presents the performance evaluation of the different layers presented in the previous chapter. On the communication layer, the goal is to evaluate the behavior of both technology and routing sublayers. On the coordination layer, it presents the performance to access tuples locally, on nearby devices and on centralized servers. Finally, in the execution and adaptation layer, it presents the coupling evaluating of CyberSupport.

5.1 Introduction

The software stack proposed in this thesis aims to help developers of self-adaptive CPS based on two premises: i) modular and uncoupled architecture, to permit the adaptation of the entire stack, to cope with the unpredictability of CPS; ii) light execution and adaptation, and to be able to handle decentralization.

5.2 Communication and Coordination

This section presents the evaluation of the USABle framework as well as the SysSU-DTS middleware, to understand the performance of both infrastructures regarding message delivery time and message loss.

5.2.1 USABle Evaluation

USABle aids developers of cyber-physical systems to implement and use communication primitives. In order to evaluate how USABle behaves in real scenarios, this section presents a performance evaluation using Android-based devices.

As mentioned in section 4.2.1, the protocols provided in the network layer are well-known. The AODV and Flooding multi-hop strategies, as well as the three epidemic-based carry-and-forward strategies, have been proposed elsewhere. Therefore, the goal is not to compare these implementations, but primarily to understand how USABle behaves under near-real-world scenarios (using real devices, but executing in an laboratory-controlled environment).

In this performance evaluation, we analyzed the implementation of five communication technologies implemented using USABle: Bluetooth, Wi-Fi, Wi-Fi Direct, SDDL and GCM. On the network layer, the goal is to evaluate the two implementations of multi-hop algorithms

(Flooding and AODV) and three epidemic-based carry-and-forward strategies.

Table 4 shows a summary of the experiments. The metrics chosen were i) latency, to evaluate the time required to send a message between devices varying the number of devices (parameter); ii) delivery rate, to understand the percentage of lost messages varying the number of devices sending messages; and iii) received messages, to evaluate how each carry-and-forward implementation delivers its messages.

Table 4 – Experiment summary

	Technology	**Routing Strat.**	**Routing Impl.**	**Metric**	**Factor**	**Parameter**
Exp 1	Bluetooth	Multi-Hop	AODV	Latency	Execution Messages	Number of Devices
Exp 2	Wifi Direct, Wi-Fi, GCM, SDDL	Multi-hop	Flooding	Latency	Number of Devices	Available Technologies
Exp 3	Bluetooth	Multi-Hop	AODV	Delivery rate	Execution Messages	Number of Devices
Exp 4	Wifi Direct, Wi-Fi, GCM, SDDL	Multi-hop	Flooding	Delivery rate	Number of Devices	Available Technologies
Exp 5	Bluetooth	Carry- and-forward	Epidemic-based	Messages received	Execution time	Available implementations
Exp 6	Bluetooth	Carry- and-forward	Epidemic-based	Messages sent/received ratio	Execution time	Available implementations

5.2.1.1 Multihop Implementations

Using the multi-hop implementations, messages were sent between devices in two different situations: i) The first situation measured the round-trip-time (RTT) for a message to reach a destination and return to the origin using Bluetooth and the AODV routing algorithm (Exp1, Table 4) and using the other technologies and the Flooding routing algorithm (Exp2, Table 4); ii) The second situation used the Flooding algorithm to analyze the number of messages lost (Exp 3 and 4, Table 4).

Situation 1. Round-trip-time. The goal of this experiment is to measure how the RTT is affected as the number of hops increases. AODV is an important baseline for multi-hop routing and was used for this experiment. For each round, 30 messages were sent. Before starting a new round, all devices were restarted and connected again. One message was sent to a destination separated by 3, 4 and 5 hops from the origin. Messages were sent with a frequency of 1 message per second and their size was around 100 bytes. The devices used in the experiments

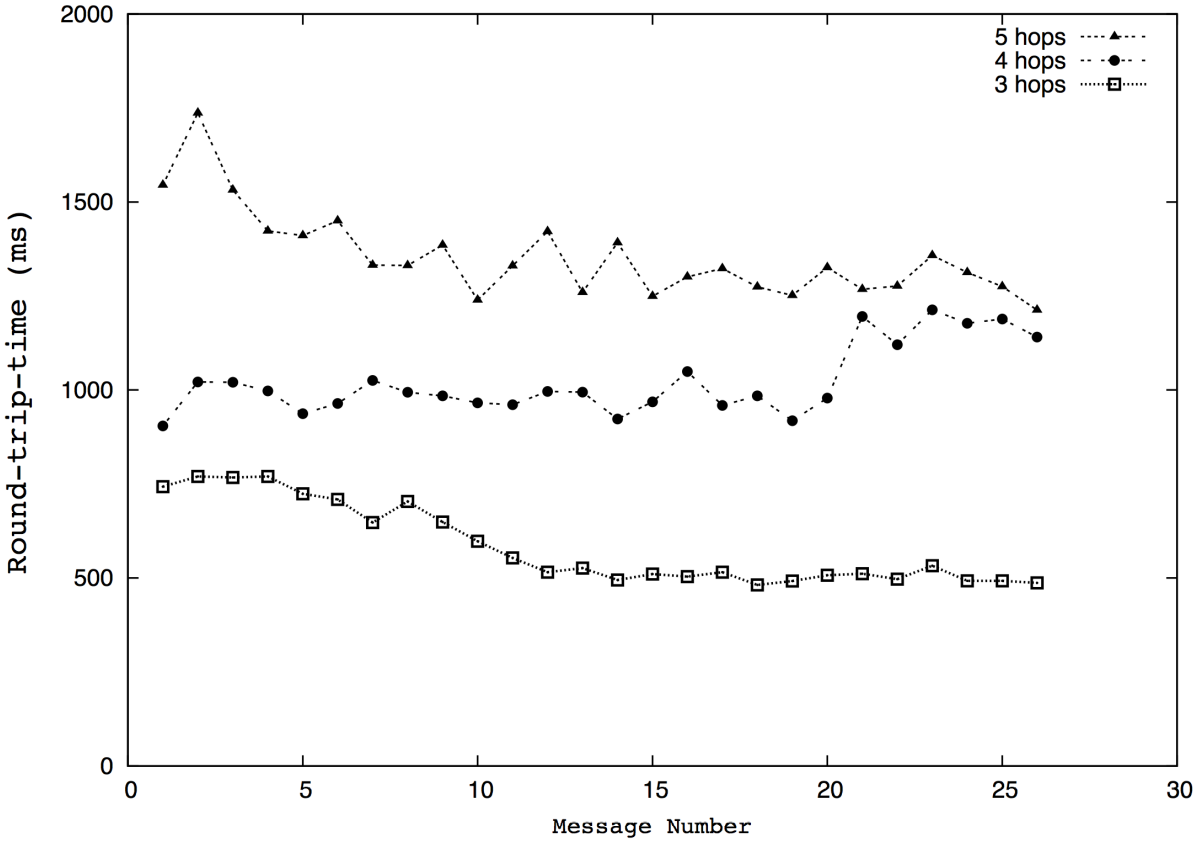

Figure 25 – RTT message delay varying the number of hops

were Sony's lt18a, xperia play, xperia x10a, xperia sola, xperia lt26i, xperia lt29i and Samsung's S3.

The average RTT for each message is shown in Figure 25. For three hops, the average RTT was around 750 ms. For each additional hop, the average RTT increased around 250 ms. The standard deviation (removed from the figure for improved readability) was around 200ms. In the beginning of the experiments, RTT tended to be a little higher and it decreases as the experiment is executed. This happens because AODV sends route request messages when a route is unknown. Once a route is known, the message is sent without querying for routes. In the end of the experiment, the RTT increased slightly, because the routes expired and another route request message was released before the actual message was sent.

That latency of 250ms per hop comprises the time to actually send the message wirelessly using Bluetooth, plus the actions executed by the framework when a message arrives in one device. The message is received by USABle from each technology module in a String format. Before each message is processed, USABle unmarshals it from *String* to a *Message* object using JSON.

In the routing algorithm, two actions are possible: i) deliver the message to an

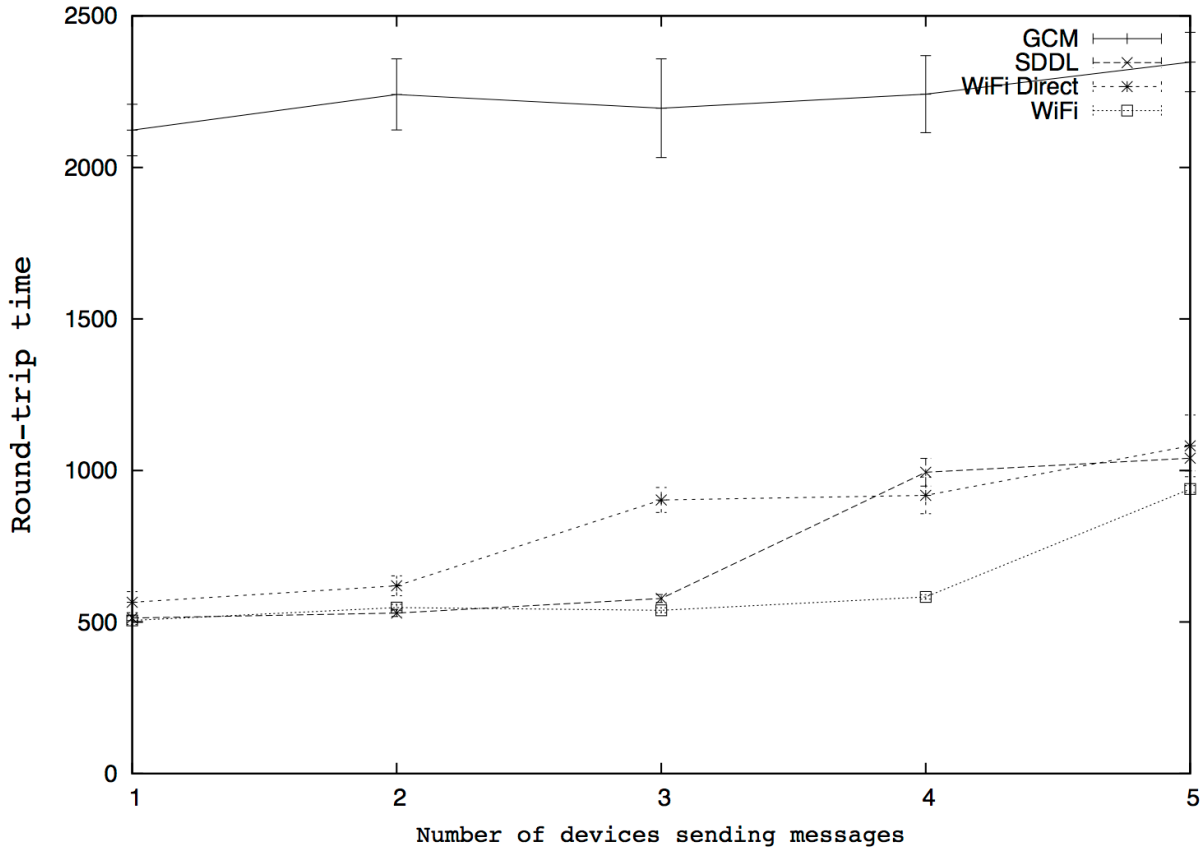

Figure 26 – Latency varying the number of devices connected

application or ii) route the message and forward it to a neighbor. Before forwarding it, the *Message* object is marshaled back to *String*. When it arrives at the destination, a reply message is sent back to the origin.

Analyzing the execution trace, the most expensive action is the unmarshaling/marshaling of JSON Strings. The use of JSON to describe the messages was a design decision to foster interoperability and facilitate the inclusion of different routing algorithms. If the framework is constructed using a single routing algorithm, the messages may be described in a lighter format.

A second experiment was executed using Wifi Direct, Wi-Fi, GCM, SDDL as the communication technologies, and using the Flooding routing strategy. The number of devices sending messages varied from 1 to 5 devices. For each experiment, 1000 messages were sent, with a frequency of 5 messages per second. Figure 26 shows the average latency time and confidence interval for sending and receiving one message to one device. The devices used in this experiment were two Google's Nexus 5 and one Nexus 4, one Samsung's S3 and one LG's G2.

Google's GCM presented the worst performance. Its architecture is based on two

servers present on the Internet: a server on Google's cloud infrastructure responsible for sending messages to/from devices and application servers. Thus, every message necessarily goes to Google's servers first, adding an extra step on the communication. The advantage of GCM was showing a small variance on the response time, when the number of devices increased. This happened because of the cloud infrastructure on the middle of the interaction, responsible for receiving and sending messages to the client and server applications.

The ping time from the local network where the devices were connected to the Google servers was approximately 110ms (round-trip min/avg/max/stddev = 105.674/110.881/118.776/3.946 ms). For every message sent, it goes from the source to Google servers, to application servers, to Google servers to the destination. To calculate the RTT time, the message goes from origin to destination and back, and eight messages are exchanged with Google servers.

The second technology for wide-area network was SDDL. By design, SDDL is built considering scalability as principle. Messages are sent using UDP as transport protocol, with message delivery guarantees implemented on top. Messages are sent between devices using a SDDL gateway and UDP. The SDDL Gateway was running on a virtual machine and hosted at the GREat lab server [1]. Ping time between the local network where the devices were connected and the SDDL gateway was approximately 38ms (round-trip min/avg/max/stddev = 37.489/38.176/40.962/0.863 ms).

Initially (1 and 2 devices sending messages), the SDDL performance was similar to local Wi-Fi and Wi-Fi direct. These two technologies use UDP as the transport protocol, but send periodic Hello messages to keep the neighbor list updated. Additionally, they do not use any centralized server and all messages are exchanged directly using mobile devices using the technology module. Since the SDDL server has significantly more processing capacity, the gain in processing time equalized the loss in the network latency. These are two reasons that explain why local messages using Wi-Fi and Wi-Fi Direct in USABle are similar to Internet-based communication using SDDL.

When the number of devices increased, the SDDL performance decreased when compared to the two Wi-Fi modules. Since more messages are being sent, the impact of the network on the message delivery time is more significant than the processing time on the device. With 4 devices, the Wi-Fi Direct performance decreased as well, since Wi-Fi Direct uses one of the devices as router within the technology. More messages means more processing on the

[1] www.great.ufc.br

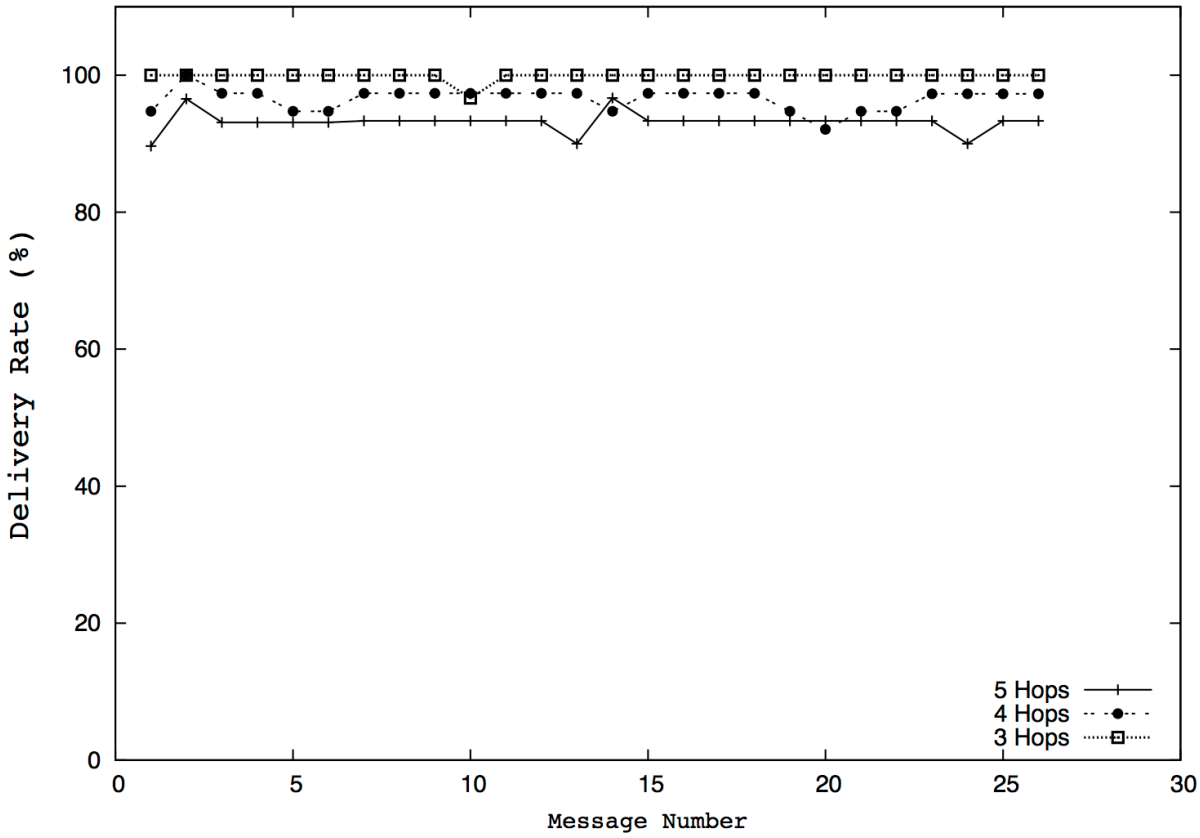

Figure 27 – Message loss varying the number of devices connected

mobile device, where on regular Wi-Fi that processing is made by the access point.

Situation 2. Message Loss. The devices were connected automatically according to the network topology algorithm implemented in each technology module. The goal was to understand how the number of lost messages varies as the number of devices increases. The Flooding algorithm was used in this experiment, since it usually sends more messages than other algorithms, stressing even more the framework. The communication technology used was Bluetooth. Figure 5.3 shows the results.

With three hops, the number of lost messages is close to zero. As the number of hops increases, the message loss is around 5% with 4 hops and 10% with 5 hops. This number is explained by two reasons: i) Flooding is an algorithm that rapidly increases the number of messages sent as the number of devices increases; and ii) the Bluetooth implementation has instability issues in some Android devices and a few messages were not delivered due to errors in the Bluetooth API inside the Android O.S. These issues caused the message to be received by the device in the lower layers of the O.S stack, but prevented the message to reach the CyberSupport module in the application layer.

When devices come into contact, they do not connect to every available device, but

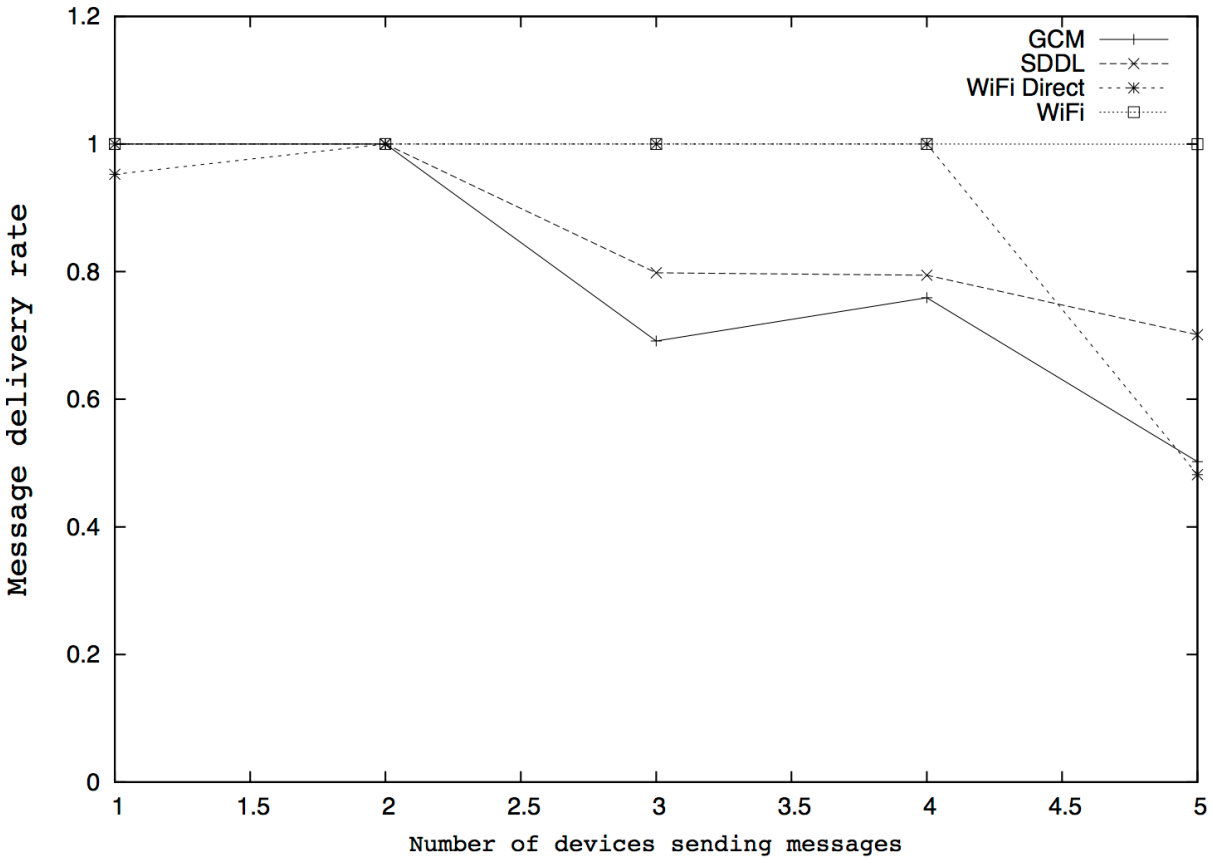

Figure 28 – Message delivery rate varying the number of devices connected

only to a subset of neighbors, according to the neighbor selection algorithm (Section 4.2.1 and (MAIA *et al.*, 2014)). Although this approach creates longer paths to devices (less links), it decreases the number of messages sent by the Flooding algorithm, minimizing the burden on the network, leading to less messages being lost.

The second experiment to evaluate the message delivery rate is presented in Figure 28. The technologies used were Wi-Fi, Wi-Fi Direct, GCM and SDDL. In this experiment, the topology chosen was to connect a new device to every other device. Since the devices communicating use Flooding, with TTL = 3, the number of messages increases exponentially with the number of devices. For instance, for one device sending messages, every message generates 2 other messages, since the message is forwarded to other devices. For 2 devices, one message generates 6 new messages. For 5 devices, 340 messages are generated.

The goal is to exhaust the technologies to understand the performance of CyberSupport, and the experiment was configured to send 2 messages per second per device.

Wi-Fi and Wi-Fi Direct delivered all messages when up to 4 devices were connected and sending messages. With 5 devices (680 messages per second) , Wi-Fi Direct decreased its delivery rate considerably (48%). Here, since one device acts as access point, along with its

activities in the operating system and USABle, some messages were lost.

SDDL with 3 devices (80 messages sent per second) and 4 devices (220 messages per second) delivered around 80% of its messages, decreasing to approximately 71% with 5 devices (680 messages per second). The SDDL Gateway was running on a virtual machine hosted in a laptop with Intel's Core i3 and 4GB of RAM. The virtual machine was configured to run with 2 virtual cores and maximum of 3GB RAM. Since all communication among devices is handled by the SDDL Gateway, improving the server configuration would improve the delivery rate.

Google's GCM showed the worst performance on the message delivery rate as well (61% for 3 devices and 49% for 5 devices). The reason here is a policy enforced by Google to queue messages arrived at Google's servers and send no more than 100 messages per second to a given device. The goal is to prevent malicious or error applications from flooding a given device, the network and their servers.

Considering the number of messages sent and the delivery rate, Flooding is not a recommended solution, since it presents the problem of message storm (sending too many messages), specially when the average number of neighbors is high.

A different implementation would improve this message loss. First, a probabilistic Flooding would decrease the number of messages sent, decreasing the load on the framework, and improving the message loss. Second, a reliable Flooding with message delivery verification could also be used.

5.2.1.2 Carry-and-Forward Implementations

The goal of the third performance evaluation is to analyze the three epidemic-based implementations. To each device, two interests out of four possible were assigned: Movies, Sports, Technology and Politics. The actual meaning of these interests is not relevant, they are just a tag describing the message content and it really could be anything else.

Every message being diffused carried a tag describing its content (Movies, Sports, Technology and Politics). Each implementation has its own heuristics to whether diffuse or not one message. The Epidemic approach always diffuses all messages it carries when it meets a new device. The Interest-based with TTL only diffuses messages with TTL above zero and to devices interested in a given message content. The Interest-based only diffuses messages with interest matching the interest of the device.

The experiment was repeated 30 times for each carry-and-forward implementation. At the beginning of each experiment, 5 messages of each interest were created in one device (Device A), in a total of 20 messages. Device A connected with Device B, exchanged messages and disconnected. Then, Device B connected and exchanged messages with Device C. Four devices were used in the experiment. The experiment considered two metrics: i) the number of messages received by one device with content that matches the interests of that device; ii) the ratio between the total number of messages sent and the number of messages received matching the interests of the receiver. The devices used in this experiment were all Samsung S3.

The metrics measured the efficiency and the cost of the carry-and-forward implementations in diffusing messages to the right devices. Figure 29 shows the number of messages received matching interests of the receiver as the experiment was executed and devices were meeting.

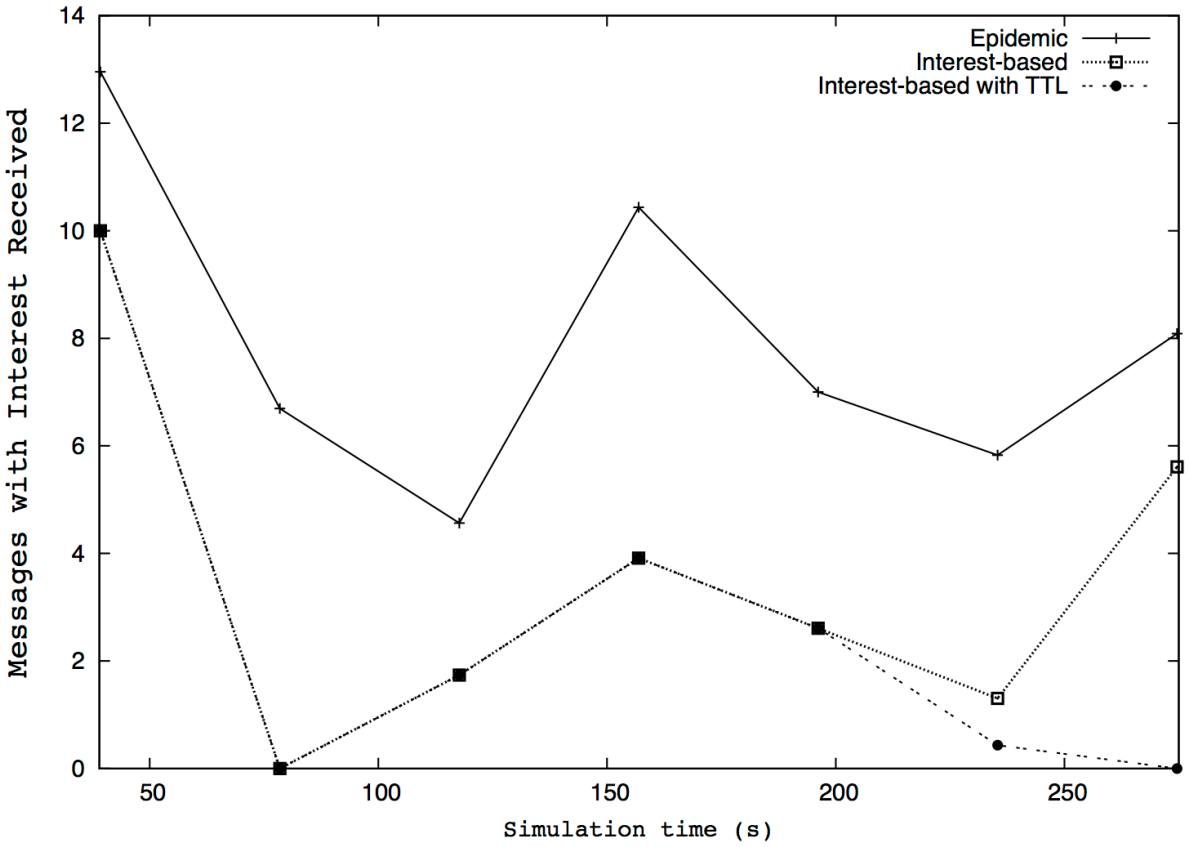

Figure 29 – Number of Interests received during the experiment

Even if a device does not have interest in a specific message content, using the Epidemic approach, it still carries and forwards messages. Thus, it was expected that the number of messages received matching the interests of the receiver would be higher using the Epidemic

approach. The two approaches that considered the interest behaved similarly until around 230 seconds, when the Interest-based with TTL decreased the number of messages being delivered (TTL expired). The peaks at approximately 30, 150 and 280 seconds were the common meeting times.

Around the time of 80 seconds, none of the approaches based on interest delivered messages. This happens because around that time, the devices that were meeting carried messages with content that did not match any of the receivers'.

Another analysis presented in Figure 30 shows the ratio between the number of messages sent and messages received matching the interests of the receiver. The Epidemic approach presented a much worse ratio than the other two approaches, since it forwards every message it carries, without considering the interests of the receivers. The two approaches that consider interest behaved similarly. Around experiment time of 230 seconds the Interest-based with TTL decreased considerably, since it stopped sending messages.

Both approaches that consider the interests of the receiver presented a ratio of 1, since every message being sent was received by an interested receiver. In the end, the TTL of the messages expired and the Interest-based with TTL stopped sending messages. The Epidemic approach presented a ratio of 2 because that is the ratio between the total number of interests (four) and the number of interests in each device (two), so for every two messages it sent, one reached an interested receiver. In the end that number increased because messages were being sent to devices that had already received that same messages.

This performance evaluation was useful for two reasons: i) it served as a proof-of-concept, where the USABle could be used to create mobile applications that exchanged messages with nearby devices; ii) it permitted to identify parts that can improved. Clearly, the use of JSON, an important architectural decision, must be reconsidered. Another important aspect is to improve the message loss. As the experiments took place, the Android O.S. eventually killed services in the Connection-aware layer responsible for sending and receiving messages. This is a feature present in the O.S. executed when the amount of available execution memory is critical.

5.2.2 SysSU-DTS Evaluation

As a proof of concept, in order to evaluate SysSU-DTS, a real experiment was performed creating a Bluetooth ad hoc network with six nodes, as well as a WiFi client-server link (everyone connected). In each node, a SysSU-DTS instance was deployed as well as a

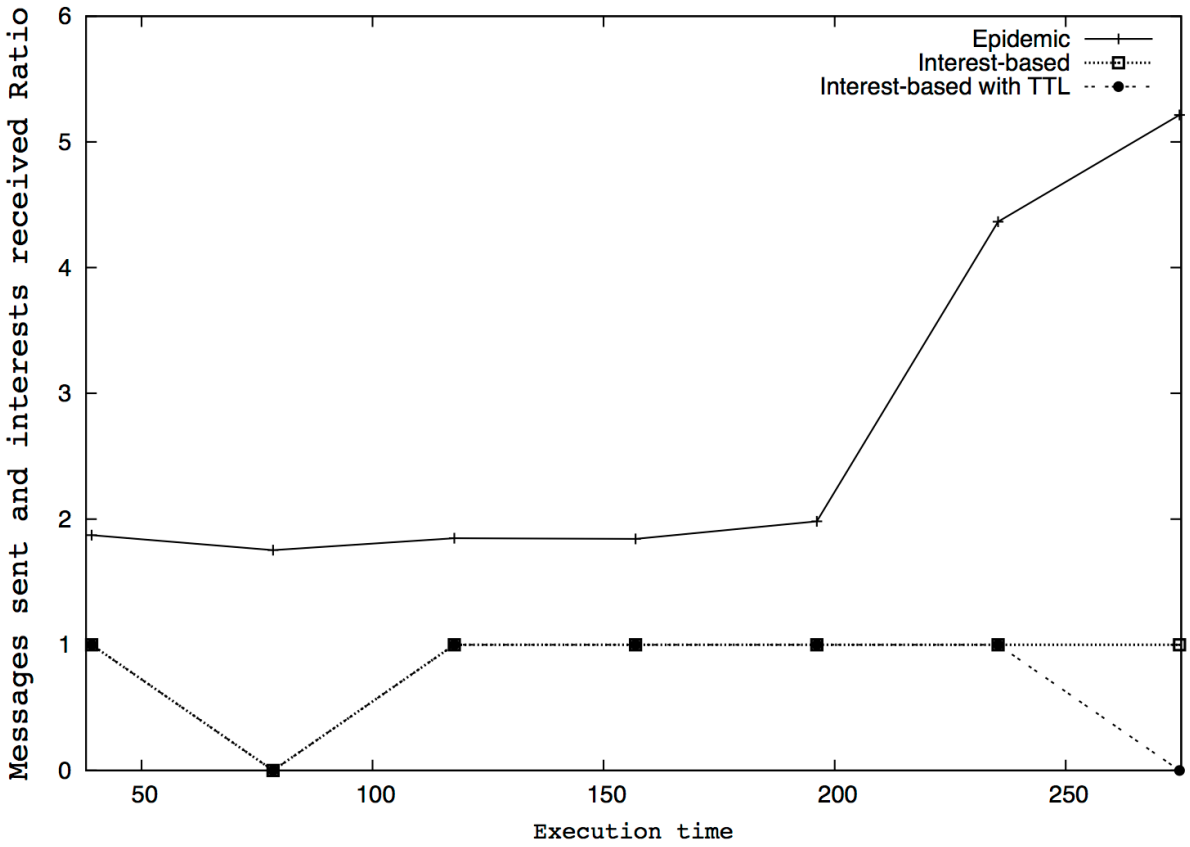

Figure 30 – Ratio between the number of interests received and the number of messages sent

mobile application built on top of SysSU-DTS. The application uses the SysSU-DTS interface to query nearby devices for context data.

One device leads the experiment execution and is responsible for triggering the workload execution, and to start a WiFi connection under a WLAN network in order to access the UbiCentre services available on the a centralized server. Furthermore, the devices were connected in a line topology, where each device has exactly one left and one right neighbor (except for the first and last devices). The main reason for that topology is to enforce multi-hop and to control the numbers of hops in each scenario.

The experiments were composed of four smartphones, two tablets and one notebook with WiFi and Bluetooth connection. The objective of this performance evaluation is to compare the response time of three different ways to access context information. The first one is query context data present in the local tuple space. Followed by tuples being queried using a centralized server, running in a notebook and listening for TCP/IP request. Finally, tuples were queried in a nearby device that is accessible from a Bluetooth connection using USABle.

The experiment set up was conducted as follows: ***System***: SysSU-DTS; ***Metric***: Response Time; ***Parameters***: Message Size (1.84 KB), access type, number of tuples in each

device (10), ad hoc network size (6 devices), number of consecutive calls (10); ***Factors***: Access type (local, server/WiFi, nearby device/Bluetooth), ad hoc network size (2, 3, 4, 5 and 6 nodes); ***Evaluation Technique***: Real system measurement; ***Workload***: A synthetic program to query context tuples present in a specific scope (great.lab1), making 10 consecutive requests with a 0.3 s interval. ***Experiment design***: One device runs the workload, executing 3 rounds for each tuple space access configuration: (i) local - internal device access, (ii) server - a WiFi access in a WLAN by TCP/IP, and (iii) nearby device - a Bluetooh access in an ad hoc network. The Bluetooth broadcast request was scaled, adding a new device per time, from 100 tuples read in 1 nearby device to 500 tuples read in 5 nearby devices.

Figure 31 shows the response time in milliseconds of ten sequential query executions. Three scenarios were considered: i) accessing tuples in a local tuple space, ii) accessing tuples in a centralized tuple space and iii) accessing tuples in a nearby device using USABle. As expected, local access is the fastest of all three scenarios. It takes around 1ms for SysSU-DTS to access local tuples. Then, accessing tuples in a centralized server took around 100 ms, followed by tuples in a nearby device, with an average response time around 230 ms.

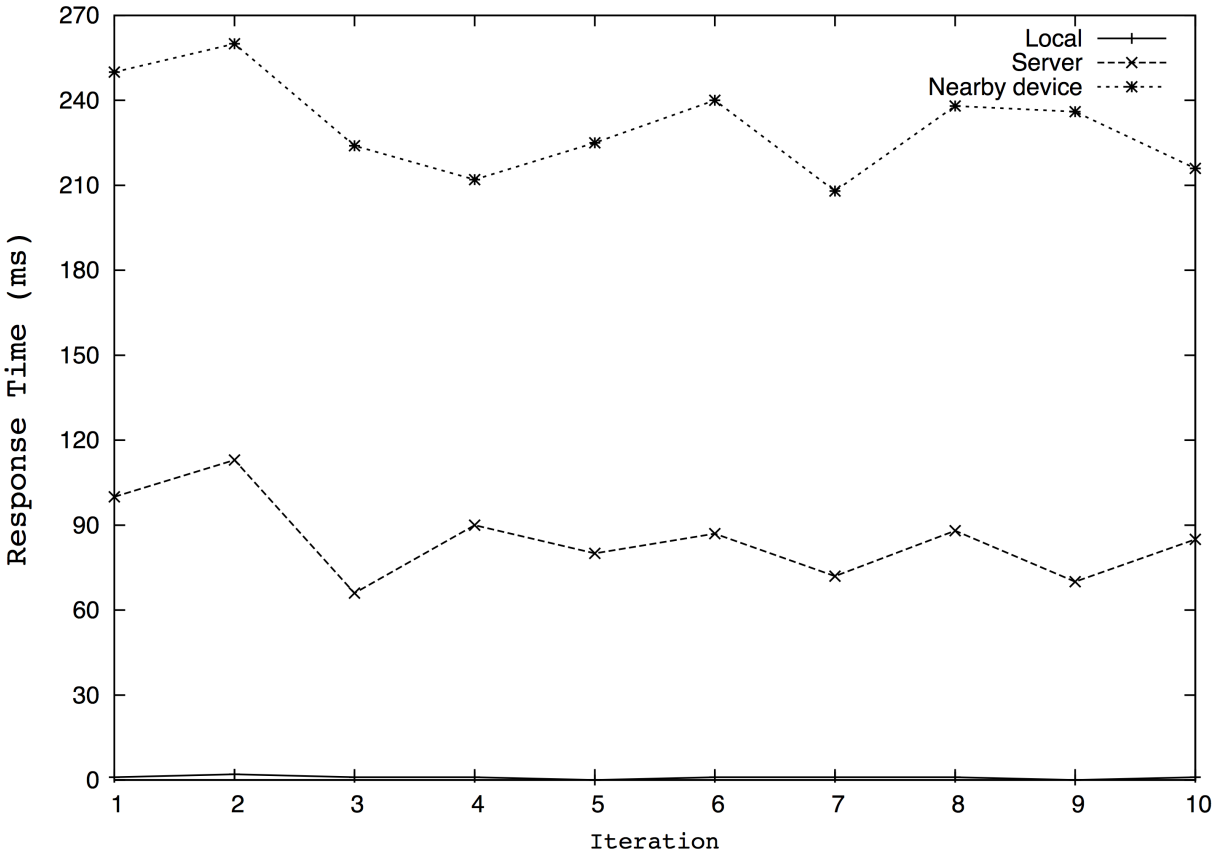

Figure 31 – Response time for local, server and nearby device.

A second experiment evaluates the system performance regarding response time in ad-hoc scenarios, increasing the number of hops, according to Figure 32. Here, whenever a new device enters the network, it begins to answer requests with 10 tuples, generating a traffic increase of 1.84 KB per device, and a 230 ms read time (sending the request and receiving the response). With five hops, the response time was 1400 ms.

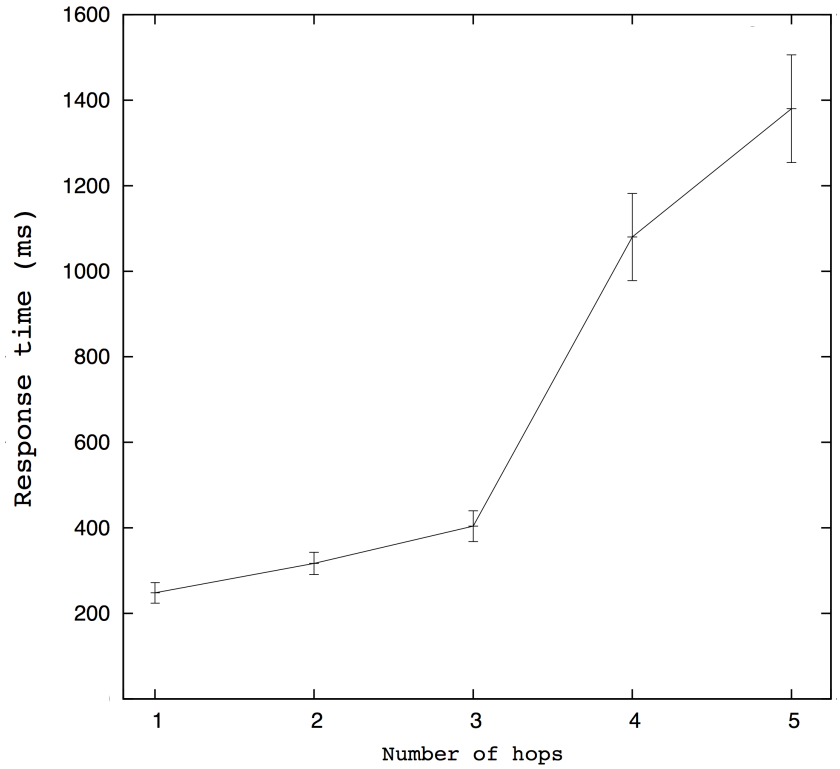

Figure 32 – Distributed tuple space broadcast query in different ad hoc network sizes

According with these experiments, the bottleneck lies on the multihop communication among distributed devices (5 devices, 1400ms). In that direction, just as an example, the LIME tuple-space (ARTAIL *et al.*, 2009) disseminated a 128KB message to 5 Pocket PCs in 1750ms. These devices were all directly connected, while here, messages were sent using multi-hop.

For scenarios with less communicating devices (up to 3 devices, around 500ms), the infrastructure behaves efficiently. Taking the GREat Tour app for instance, the available nearby users CAC sent broadcast messages to nearby devices with a TTL equals three. Thus, every device within 3 hops would respond to the query. To restrict request messages within environments, nearby users requests carried the room description. Only devices present in the room described by the message request responded. In that scenario, according with Figure 32, it would take around 500ms to receive the responses.

Table 5 – Applications developed with CyberSupport and the created CACs

GREat Tour	Indoor Location: QR-Code; NFC Tags. Users and Items: Server-based; Nearby users; User Information; List Environment Items. Device: Execution Memory; Storage Memory; Battery; Network; SoundLevel. Environment: Login; Download Photo/Video; GPSLocation
FAlert	FallDetection, SendOutMessages
GREat Mute	UserAgenda, GPSLocation, SoundMode

5.3 Execution and Adaptation

The development of self-adaptive CPS applications using CyberSupport is based primarily on the use of components to encapsulate access to sensors and actuators. The interaction between application and components is based on a tuple space, aiming to improve system modularity and to facilitate adaptation. Here, the interaction using tuple spaces is based on the assumption that it minimizes coupling among interacting entities.

To understand how CyberSupport helps to create CPS applications with lower coupling levels, three pervasive applications previously developed by researchers at GREat were refactored to use the functionalities present in CyberSupport (Section 4.4). Before the refactoring was carried out, all three applications were constructed considering a coordination model based on direct method invocation.

The GREat Tour app was the first application refactored. The other two applications were i) FAlert, an Android application that detects when the user falls and sends out help messages; and ii) GREat Mute, an Android application that access the user's calendar to put the device in silent mode whenever a meeting begins. Table 5 shows these three applications and the CACs developed for each one of them.

These two versions of each application, one with and another without CyberSupport, were used to compare their coupling levels, analyzed according to two different perspectives: i) system design, measuring the strength between system modules according to software engineering metrics; and ii) system execution, analyzing how coupled the interacting modules are at runtime.

5.3.1 System Design Coupling

System design coupling was measured using three coupling metrics (SANTANNA *et al.*, 2007), namely: i) Concern diffusion over components (CDC), that counts the number of

classes and interfaces implementing a given functionality, along with the number of other classes and interfaces accessing the classes and interfaces that implements that functionality; ii) Concern diffusion over operations (CDO), which counts the number of methods implementing a given functionality, along with the number of times other methods access the methods that implement that functionality; iii) Coupling Between Components (CBC), which counts the number of times two components are linked, by accessing their methods. Additionally, one last metric was used to show the size of the two systems, considering the number of code lines. A more detailed discussion on coupling among system modules is presented in Appendix A. Figure 33 shows the three system design coupling metrics, along with the Lines of Code (LoC) metric, without and with CyberSupport.

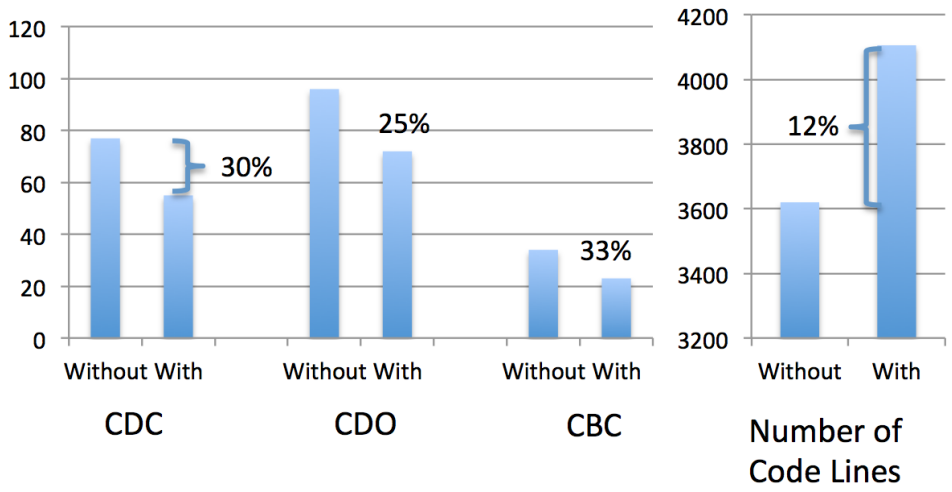

Figure 33 – System design coupling metrics with and without CyberSupport

These three coupling metrics measure the strength with which the application modules are connected only with the components responsible for implementing the functionalities to access sensors and actuators presented in Table 5. The connection strength among the rest of the system is not considered, since it would measure the coupling of the overall system, and that depends basically on how well the system was designed.

Considering all three coupling metrics, with CyberSupport the system modules were less coupled, or more uncoupled. The main reason for that was the use of the get/set interfaces presented in Algorithms 4.6, 4.7 and 4.8. They reduce the interaction with each functionality in Table 5 to one interface invocation for each component.

Another benefit of CyberSupport regarding system design coupling is because two modules are connected based on their signature (input/output parameters and description based on the CIB). The actual implementation is defined at runtime by LoCCAM based on available

implementations. Signature coupling is the most desirable type of coupling (Refer to Appendix A).

The use of tuple space as means to permit the interaction between application and CACs is a major contribution of CyberSupport. It facilitates self-adaptation driven by the Analyze and Plan phases of the MAPE-K loop, since the interaction infrastructure (get/set methods and SysSU) are kept unchanged, but the actual CAC execution is defined by the CIB concept. Changing the CAC requires only to change the CIB to be used.

The number of code lines has increased when using CyberSupport. That happened because for each get/set invocation, the programmer has to implement a Pattern (Algorithm 4.1), a Filter (Algorithm 4.2) and/or a Reaction (Algorithm 4.3). When designing the interaction mechanisms present on CyberSupport, to improve uncoupling among system modules was the primary concern, at the cost of applications with more line codes.

5.3.2 System Execution Coupling

CPS applications are formed by distributed modules interacting using available communication protocols (see sections 2.4.2 and 3.2). In that scenario, unpredictability is an important challenge to be considered by developers. To create CPS applications capable of handling unpredicted situations, the interacting modules must be uncoupled at runtime according to three dimensions: i) address, where interacting modules must be able to interact without relying on each other's addresses; ii) time, where interacting modules must be able to interact without being available at the same time; iii) synchrony, where interacting modules must be able to interact asynchronously.

Considering these dimensions, three system execution coupling metrics are proposed by this doctoral thesis (refer to Appendix A) namely: i) Number of Address References (NAR), which counts the number of times two modules interact based on a network/application address; ii) Number of Online Interactions (NOI), which counts the number of times two modules must be online at the same time for an interaction to take place; and iii) Number of Synchronous Interactions, which counts the number of times a synchronous interaction takes place.

These three metrics give an indication of how coupled two modules are at runtime. Higher coupling means less resiliency to failures, since the overall system only behaves normally when all modules are available. For instance, by relying on a network address (NAR metric) or on the availability of a module at a given time (NOI metric), the system becomes less prone to

behave normally when facing disconnections.

Using the NAR, NOI and NSI metrics, the three applications (GREat Tour app, FAlert and GREat Mute) considered in the previous section were scrutinized. The goal is to understand how CyberSupport improves the uncoupling among execution modules. Figure 34 shows each execution uncoupling metric without and with CyberSupport.

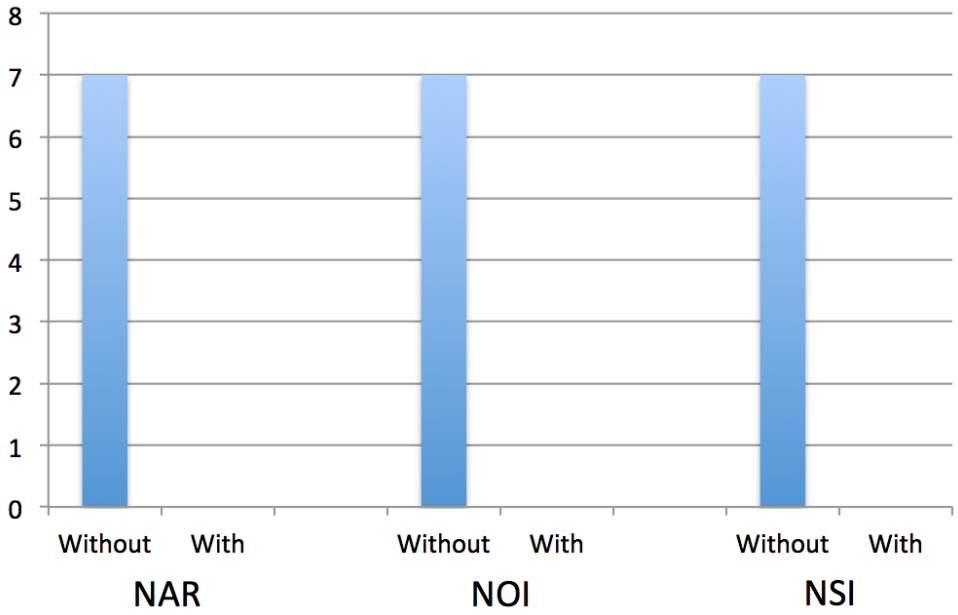

Figure 34 – System execution coupling metrics with and without CyberSupport

Without CyberSupport, the interaction mechanism these applications were using is based on direct or remote method invocation, which is exactly the type of interaction the three execution coupling metrics address and CyberSupport aims to minimize.

With CyberSupport, the values for the three execution coupling metrics is zero, for basically two reasons. First, any address references (NAR) is either encapsulated within the CAC, and the application has no access to that address, and because coordination takes place based on a distributed tuple space (SysSU-DTS) and any message exchange is carried out based on local spaces, where the infrastructure translates network/application addresses into application messages. Second, time (NOI) and synchrony (NSI) uncoupling is accomplished using the get/set asynchronous interfaces and the use of the Reaction mechanism (Algorithm 4.7).

5.4 Conclusions

This chapter presented the performance evaluation of the entire software stack executed using real devices. In the communication layer, it analyzed five communication tech-

nologies, two multi-hop algorithms and three carry-and-forward implementations, considering the round trip time and message loss metrics. Another experiment on the coordination layer showed the time to access local tuples, and tuples on nearby devices and on centralized servers.

The evaluation on the execution and adaptation layer was carried out by refactoring three existing applications to use CyberSupport. The idea was to leverage how uncoupled the new version of each application was considering two dimensions. First, System Design Coupling measured the connection strength among system modules according to three metrics (CDC, CDO and CBC). It showed an improvement around 30%, due to the use of common interfaces, coupling based on signature and the use of tuple spaces. Second, System Execution Coupling measured the connection strength among execution modules. Here, CyberSupport removed completely the address, time and synchrony couplings, mainly because it used CACs and the tuple space to handle addresses and the use of Pub/Sub mechanisms with tuple spaces to implement asynchrony.

6 CONCLUDING REMARKS AND FUTURE WORK

This chapter describes the final remarks regarding CyberSupport: System support for decentralized self-adaptive cyber-physical systems. Section 6.1 presents the contributions and results. In section 6.2, a discussion on the hypothesis and research questions is presented. Finally, section 6.3 presents the future work.

6.1 Results and Contributions

The main result of this thesis is a decentralized and uncoupled support infrastructure to aid developers to implement self-adaptive cyber-physical systems called CyberSupport. To handle the different requirements and challenges of self-adaptive CPS, the software stack is divided in two layers: i) Communication and coordination; and ii) Execution and adaptation.

The communication layer offered mechanisms to let developers exchange and route messages between distributed devices, without paying too much attention to technologies and routing activities, and providing management interfaces to adapt the entire communication infrastructure. Among the analyzed related work, USABle was the only one to support various communication technologies and to permit the development/deployment of different routing algorithms on top of available technologies.

The following results can be highlighted in the communication layer:

- development of five communication technologies: Bluetooth, Wi-Fi, Wi-Fi Direct, GCM and SDDL;
- development of two multihop routing strategies (AODV and Flooding) and three carry-and-forward strategies (Epidemic, Epidemic with Interest and Epidemic with Interest and TTL);

The coordination layer is constructed based on **decentralized** tuple spaces. Reading and writing operations are executed first on local tuple spaces, then on tuple spaces present on nearby devices and on servers present on the Internet. On top of the tuple space, a pub/sub mechanism offers filter and reactions primitives to implement response to environment events.

Uncoupling is improved in two different moments: i) uncoupled interactions between distributed devices and ii) uncoupled interaction between applications and components responsible for accessing and controlling environment resources. While most of related work use tuple spaces for interaction among devices, the interaction between application and components

is coupled in at least in one of the three categories: time, address or synchrony. Here, the use of tuple space to permit the interaction of application and components improved the design and execution coupling.

The following results can be highlighted in the coordination layer:

- distributed tuple space infrastructure to permit the
  - interaction among distributed devices;
  - interaction among application and components;

Using the underlying communication and coordination layer, the execution and adaptation layer permits the specification of application behavior based on a set of actions, which is defined by the description of components to be used, and reactions, to specify specific behavior in response to environment events. Each action is specified using a component model and executed using components present on decentralized devices. As a contribution, it improves uncoupling among applications, components and devices.

The following results can be highlighted in the execution and adaptation layer:

- Adaptable component infrastructure to access sensors and actuators
  - adaptation based on CIB description;
  - adaptation based on reactions;

6.1.1 CyberSupport Compared to the Related Work

CyberSupport is a supporting infrastructure for self-adaptive CPS, with three major contributions: i) a modular and adaptable communication infrastructure; ii) a decentralized coordination infrastructure, fostering uncoupled interactions between distributed devices; and iii) an adaptable component management infrastructure to handle the access to sensors and actuators present in the environment.

The related work are now presented in tables 6 and 7 with the addition of CyberSupport.

Interoperability is considered in most of the related work (except (JIAO; SUN, 2013)) by using service-orientation. In CyberSupport, the CIB concept is used to describe resources, but leaving to developers the decision of which data representation to use (SOAP, REST, Ontologies). CIB was created considering lightness and generality, but can be mapped at runtime to other formalisms.

Scalability is improved when decentralization and lightness are present. Here, the

Table 6 – CyberSupport compared to the related work

	Interoperability	Scalability	Extensibility	Planning
Link Smart (ZHANG *et al.*, 2014a)	✓	✗	✓	✓
3PC (HANDTE *et al.*, 2012)	✓	✗	✓	✓
fraSCAti (SEINTURIER *et al.*, 2012)	✓	✓	✓	✗
MUSIC (FLOCH *et al.*, 2013)	✓	✗	✓	✓
Jiao (JIAO; SUN, 2013)	✗	✓	✓	✓
CAMPUS (WEI; CHAN, 2013)	✓	✗	✓	✓
CyberSupport	✓	✓	✓	✗

decision between centralized or decentralized architecture is left to developers to implement it, but offering mechanisms to foster decentralization. Additionally, lightness is considered in the use of CIB concepts, where the description of the functionality is easily derived from the tree of concepts. Solutions using ontologies or based on a single centralized entity do not scale well, which is the case for Link Smart, 3PC, MUSIC and CAMPUS. CyberSupport is based on a light formalism (CIB) and gives suport to local/uncoupled interactions.

Planning is not implemented by this doctoral thesis and is left as future work.

Support for CPS communication is the first contribution of CyberSupport, dealing with the requirements presented in section 3.1 (leveraging proximity and mobility, exploring available communication technologies and runtime management). Among the related work, 3PC handles these requirements and Link Smart facilitates the runtime management of communication layer.

Uncoupled coordination is a direct consequence of shared spaces, which is a relevant design decision of this work. MUSIC also uses shared spaces, but only as a service directory. MUSIC and fraSCAti accomplish uncoupling by using connectors that can be changed at runtime. Although it improves uncoupling (referential and synchronization), it still presents the downside of online interactions. The same is valid for 3PC.

Considering the abstractions to describe discover and access, CyberSupport uses a simple mechanism to describe resources based on the CIB concept of concepts, and the current implementation for the CAC discovery is based on a simple centralized repository. An extension

Table 7 – Positioning CyberSupport considering CPS requirements

	Commu-cation	Coordi-nation	Sensor/App Separation	Openness	Abstractions
Link Smart (ZHANG *et al.*, 2014a)	✓	✗	✓	✓	✓
3PC (HANDTE *et al.*, 2012)	✓	✗	✓	✓	✓
fraSCAti (SEIN-TURIER *et al.*, 2012)	✗	✗	✓	✓	✗
MUSIC (FLOCH *et al.*, 2013)	✗	✗	✓	✓	✓
Jiao (JIAO; SUN, 2013)	✗	✗	✓	✗	✓
CAMPUS (WEI; CHAN, 2013)	✗	✗	✓	✓	✓
CyberSupport	✓	✓	✓	✓	✓

to CyberSupport to fully support this requirement is under development and is described as a future work.

6.1.2 Publications

This section describes the publications generated as a direct or indirect result of this doctoral thesis. Table 8 shows the publications as a direct result of the contribution described in this doctoral thesis.

In the communication layer, two papers were written. The first (Number 1) is called USABle – A Communication Framework for Ubiquitous Systems and was published in the Advanced Information Networking and Applications (AINA), describing the communication infrastructure. An extension of this paper, including more communication technologies, describing more with more details the infrastructure and with a broader performance evaluation was submitted to the Journal of Systems and Software and is currently under review.

In the coordination layer, a paper called A coordination framework for dynamic adaptation in ubiquitous systems based on distributed tuple space, published in the International Wireless Communications and Mobile Computing Conference, as a partnership with a master

student in our research group, describes and evaluates the distributed tuple space module.

In the execution and adaptation layer, two papers described the component model and its interaction with the tuple space. The first paper called LOCCAM - loosely coupled context acquisition middleware, published in the Symposium on Applied Computing, presented the component model and CIB concepts. The second, called An Adaptive Context Acquisition Framework to Support Mobile Spatial and Context-Aware Applications published in the International Symposium on Web and Wireless Geographical Information Systems, as a partnership with a master student in our research group, used the component model to implement spatial filters for mobile applications.

Table 8 – Publications as a direct consequence of this doctoral thesis

No.	Citation	Type
1	Maia, Marcio E.F. ; ANDRADE, R. M. C. ; Carlos A.B. de Q. Filho ; BRAGA, R. ; Aguiar, Saulo ; Bruno Gois Mateus ; CASTRO, RUTE N. S. ; Fredrik Toorn . USABle – A Communication Framework for Ubiquitous Systems. In: Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on, 2014, Victoria, BC. 2014 IEEE 28th International Conference on Advanced Information Networking and Applications (AINA), 2014. p. 81-88.	Conference
2	NETO, B. J. A. ; ANDRADE, R. M. C. ; FONTELES, A. S. ; MAIA, Marcio E. F. ; Viana, Windson . A coordination framework for dynamic adaptation in ubiquitous systems based on distributed tuple space. In: 9th International Wireless Communications and Mobile Computing Conference, 2013, Cagliari. Proceedings of the 9th International Wireless Communications and Mobile Computing Conference, 2013. p. 1430-1435.	Conference
3	MAIA, Marcio E. F. ; GADELHA, ROMULO ; FONTELES, ANDRE ; NETO, BENEDITO ; Viana, Windson ; ANDRADE, ROSSANA M. C. . LOCCAM - loosely coupled context acquisition middleware. In: the 28th Annual ACM Symposium, 2013, Coimbra. Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13. New York: ACM Press, 2013. v. 1. p. 534-541.	Conference
4	FONTELES, B. J. A. ; NETO, B. J. A. ; MAIA, Marcio E. F. ; ANDRADE, R. M. C. ; Viana, Windson . An Adaptive Context Acquisition Framework to Support Mobile Spatial and Context-Aware Applications. In: 12th International Symposium on Web and Wireless Geographical Information Systems, 2013, Alberta. Procedings of the 12th International Symposium on Web and Wireless Geographical Information Systems,, 2013. v. 7820. p. 100-116.	Conference

During the development of this doctoral thesis, other papers were published, not directly related to the contribution here presented, but somehow contributing to the acquired knowledge and ideas that led to the design presented in the previous chapters. Table 9 shows the papers publish during the construction of this doctoral thesis.

Table 9 – Secondary publications that helped to construct the current solution

No.	Citation	Type
5	Marcio E. F. ; ROCHA, L. S. ; MAIA, P. H. M. ; ANDRADE, R. M. C. . An Autonomous Middleware Model for Essential Services in Distributed Mobile Applications. In: Venkatasubramanian, N., Getov, V., Steglich, S. (eds.) LNICST 93, 2012, Heidelberg. 4th International ICST Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications. Heidelberg: Springer, 2012. p. 57-70	Conference
6	MAIA, Marcio E. F. ; FILHO, JOAO BOSCO F. ; DE Q. FILHO, CARLOS A. B. ; CASTRO, RUTE N. S. ; ANDRADE, ROSSANA M. C. ; TOORN, FREDRIK . Framework for building intelligent mobile social applications. In: the 27th Annual ACM Symposium, 2012, Trento. Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12. New York: ACM Press. p. 525	Conference
7	Rocha, Lincoln S. ; F., J. BOSCO FERREIRA ; LIMA, FRANCISCO F. P. ; MAIA, Marcio E. F. ; Viana, Windson ; CASTRO, MIGUEL F. DE ; ANDRADE, ROSSANA M. C. . Ubiquitous Software Engineering: Achievements, Challenges and Beyond. In: 2011 25th Brazilian Symposium on Software Engineering (SBES), 2011, Sao Paulo. 2011 25th Brazilian Symposium on Software Engineering, 2011. p. 132	Conference
8	Marinho, Fabiana G. ; Andrade, Rossana M.C. ; Werner, Claudia ; Viana, Windson ; Maia, Marcio E.F. ; Rocha, Lincoln S. ; Teixeira, Eldanae ; Filho, Joao B. Ferreira ; Dantas, Valeria L.L. ; Lima, Fabricio ; Aguiar, Saulo . MobiLine: A nested software product line for the domain of mobile and context-aware applications. Science of Computer Programming (Print), v. 77, p. 1-18, 2012	Journal
9	Marcal, E. ; MAIA, M ; VIANA, W ; C., R. M. . Geomovel: Um Aplicativo para Auxilio a Aulas de Campo de Geologia. In: Simposio Brasileiro de Informmtica na Educacao - SBIE, 2013, Campinas. Anais do XXIV Simposio Brasileiro de Informatica na Educacao - SBIE, 2013	Conference

6.2 Hypothesis and Research Questions Analysis

Section 1.3 hypothesized that the development of self-adaptive cyber-physical systems should use an uncoupled and decentralized execution infrastructure. According to the literature and the coupling evaluation presented in sections 5.3.1 and 5.3.2, the use of CyberSupport improved system design and system execution coupling. This happened because all interactions between systems, components and resources are executed based on tuple space and pub/sub mechanisms. Finally, self-adaptation permits application to adapt is behavior and structure to changes detected in the environment. Support for uncoupled and decentralized self-adaptation in CPS is the main contribution of this thesis.

The research questions presented in section 1.3 are discussed as follows:

Research Question 1: What are the main requirements to implement decentralized self-adaptive cyber-physical systems?

Decentralized self-adaptive CPS are based on the interaction between distributed entities. Chapter 2 discusses the requirements for implementing decentralized self-adaptation, along with the development of CPS.

Sections 2.2 and 2.3 discusses the requirements for decentralized self-adaptation, presenting a reference architecture based on the MAPE-K paradigm and discussing existing patterns for decentralized self-adaptation. From section 2.2, which presents the MAPE-K divided in four phases, this thesis implemented the monitor and execute phases using components responsible for accessing and controlling environment resources. The analyze and plan phases are left as future work.

Section 2.4 presents the requirements for developing CPS, dividing them in interaction and communication, along with resource management. The contributions of this thesis are the following: i) interaction is accomplished based on message exchange through wireless communication interfaces. ii) coordination based on decentralized tuple spaces; and iii) execution of CPS based on an uncoupled infrastructure, letting developers to specify the overall goals and actions, whereas the infrastructure is responsible for managing and adapting its behavior.

Research Question 2: What are the main challenges to implement a middleware layer for self-adaptive cyber-physical systems?

According to section 1.1, the challenges are i) decentralization, ii) mobility, iii) uncoupled spontaneous interaction; iv) interoperability and v) context awareness and adaptability.

Decentralization in self-adaptive systems was illustrated in sections 2.3 and considered throughout the design of the contribution. In the communication and coordination layer (section 4.2), technologies based on direct connection, routing algorithms for decentralized mobile environments and tuple space and pub/sub-based solutions permitted the creation of decentralized solutions.

Mobility is considered in the communication layer, using communication technologies able to handle communication over wide area networks along with technologies that permit direct connection. Also, because of routing algorithms capable of sending messages when fixed infrastructures are not available. Finally, uncoupled interaction is accomplished using distributed tuple spaces, where all interactions are coordinated over available decentralized tuple spaces.

In section 4.3., interoperability is partially solved by the use of CIB. Partially because interacting devices must be based on the formalism presented in subsections 4.3.1 and 4.3.2. Although this is a light formalism that solves the interoperability problem from the perspective

of resource description, the final solution for the interoperability problem is left untouched as future work.

Context-awareness is implemented using components to implement access to sensors (CACs) or context components capable of making complex inferences, along with the filters and reaction mechanism presented in section 4.3.3.

Research Question 3: How coupling affects the development of self-adaptive CPS?

Uncoupled systems are more easily maintainable and evolvable. Since self-adaptation is the act of analyzing the execution state of a system and performing maintenance activities at runtime, coupling is an mandatory quality attribute.

In the communication layer, uncoupling is achieved using a layer-based architecture, offering different communication modules and routing algorithms, all interacting based on immutable interfaces. Maintenance is accomplished by changing modules, but keeping their interfaces, respecting contracts (interfaces) defined at design-time.

In the execution and adaptation layer, uncoupled components, each encapsulating a single responsibility foster the overall modularity of the system. Changing system behavior is executed by changing the corresponding component.

Uncoupling is present because all interactions are necessary executed using a tuple space with pub/sub functionalities. Thus, the interaction between components and systems, both local and remote, is separated in time, address and synchronization.

6.3 Future Work

Following up the work developed during this doctoral thesis, here is a list of potential future work to be developed:

- **Development of execution plan for CPS.** An execution plan is the sequence of actions generated by the Planning phase of the MAPE-K loop. These actions are generated to make the system move from an initial state to a final desirable goal. They permit the system to handle unpredicted situations and evolving goals. These are two characteristics desired from self-adaptive CPS.
- **Foster interoperability in CPS**. The number and heterogeneity of devices forming CPS require novel solutions to permit the construction and execution of systems, without relying on human configuration and management. The use of self-adaptation to specify and manage these interactions, aiming to support interoperability is a promising concept.

- **Description and discovery of CPS resources.** Although the solution presented in this doctoral thesis handles description and discovery of components to interact with CPS devices, they can be improved to handle other aspects of CPS execution, such as the description of pre- and post- conditions and invariants. Additionally, the decentralized discovery of resources is mandatory.
- **Exception handling mechanism for CPS.** The execution of actions to sense and control the environment must be supported by exception handling mechanisms to improve system resiliency. Up to this point, the CAC execution using get/set methods returned a status code. That approach must be extended to support the specification of execution and environment exceptions.

BIBLIOGRAPHY

ALFÉREZ, G. H.; PELECHANO, V. Dynamic evolution of context-aware systems with models at runtime. In: **Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems**. Berlin, Heidelberg: Springer-Verlag, 2012. (MODELS'12), p. 70–86. ISBN 978-3-642-33665-2.

ALLIANCE, O. **OSGi service platform, release 3**. s.l.: IOS Press, Inc., 2003.

ALLJOYN. **AllJoyn**. 2013. Disponível em www.alljoyn.org; Online; Acessado em 18 de Outubro de 2013.

ANTSAKLIS, P. Goals and challenges in cyber-physical systems research editorial of the editor in chief. **Automatic Control, IEEE Transactions on**, IEEE Computer Society, Washington, DC, USA, v. 59, n. 12, p. 3117–3119, Dec 2014. ISSN 0018-9286.

ARTAIL, H.; AL-HALABI, F.; CHEHAB, A. The design and implementation of an ad hoc network of mobile devices using the lime ii tuple-space framework. **Wireless Communications, IEEE**, IEEE, Washington, DC, USA, v. 16, n. 3, p. 52–59, 2009.

AUTILI, M.; INVERARDI, P.; PELLICCIONE, P.; TIVOLI, M. Developing highly complex distributed systems: a software engineering perspective. **Journal of Internet Services and Applications**, Springer-Verlag, Berlin, Heidelberg, v. 3, n. 1, p. 15–22, 2012. ISSN 1867-4828.

BARESI, L.; GHEZZI, C. The disappearing boundary between development-time and run-time. In: **Proceedings of the FSE/SDP workshop on Future of software engineering research**. New York, NY, USA: ACM, 2010. (FoSER '10), p. 17–22. ISBN 978-1-4503-0427-6.

BECKER, C.; HANDTE, M.; SCHIELE, G.; ROTHERMEL, K. Pcom - a component system for pervasive computing. In: **Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)**. Washington, DC, USA: IEEE Computer Society, 2004. (PERCOM '04), p. 67–. ISBN 0-7695-2090-1.

BENCOMO, N. On the use of software models during software execution. In: **Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (MISE '09), p. 62–67. ISBN 978-1-4244-3722-1.

BLAIR, G.; BENCOMO, N.; FRANCE, R. Models@run.time. **ACM Computer**, ACM, New York, NY, USA, v. 42, n. 10, p. 22–27, Oct 2009. ISSN 0018-9162.

BLAIR, G. S.; BENNACEUR, A.; GEORGANTAS, N.; GRACE, P.; ISSARNY, V.; NUNDLOLL, V.; PAOLUCCI, M. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In: **Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware**. Berlin, Heidelberg: Springer-Verlag, 2011. (Middleware'11), p. 410–430. ISBN 978-3-642-25820-6.

BORGIA, E. The internet of things vision: Key features, applications and open issues. **Computer Communications**, IEEE Computer Society, Washington, DC, USA, v. 54, p. 1 – 31, 2014. ISSN 0140-3664.

BRAGA, R. B.; TAHIR, A.; BERTOLOTTO, M.; MARTIN, H. Clustering user trajectories to find patterns for social interaction applications. In: **Proceedings of the 11th international**

conference on Web and Wireless Geographical Information Systems. Berlin, Heidelberg: Springer-Verlag, 2012. (W2GIS'12), p. 82–97. ISBN 978-3-642-29246-0.

BRANDAO, R.; FRANCA, P.; MEDEIROS, A.; PORTELLA, F.; CERQUEIRA, R. The cas project: A general infrastructure for pervasive capture and access systems. In: **Proceedings of the 28th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2013. (SAC '13), p. 975–980. ISBN 978-1-4503-1656-9.

BROMBERG, Y.-D.; GRACE, P.; REVEILLERE, L. Starlink: Runtime interoperability between heterogeneous middleware protocols. In: **ICDCS**. Washington, DC, USA: IEEE Computer Society, 2011. p. 446–455. ISBN 978-0-7695-4364-2.

BRUMITT, B.; MEYERS, B.; KRUMM, J.; KERN, A.; SHAFER, S. A. Easyliving: Technologies for intelligent environments. In: **Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing**. London, UK: Springer-Verlag, 2000. (HUC '00), p. 12–29. ISBN 3-540-41093-7.

BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUÉMA, V.; STEFANI, J.-B. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 36, n. 11-12, p. 1257–1284, set. 2006. ISSN 0038-0644.

BURES, T.; GEROSTATHOPOULOS, I.; HNETYNKA, P.; KEZNIKL, J.; KIT, M.; PLASIL, F. Deeco: An ensemble-based component system. In: **Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering**. New York, NY, USA: ACM, 2013. (CBSE '13), p. 81–90. ISBN 978-1-4503-2122-8.

BURROWS, R.; GARCIA, A.; TAIANI, F. Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. In: MACIASZEK, L.; GONZALEZ-PEREZ, C.; JABLONSKI, S. (Ed.). **Evaluation of Novel Approaches to Software Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, (Communications in Computer and Information Science, v. 69). p. 277–290. ISBN 978-3-642-14818-7.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern-oriented Software Architecture: A System of Patterns**. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN 0-471-95869-7.

CABRI, G.; FERRATI, L.; LEONARDI, L.; MAMEI, M.; ZAMBONELLI, F. Uncoupling coordination: Tuple-based models for mobility. In: BELLAVISTA, P.; CONRRADI, A. (Ed.). **The Handbook of Mobile Middleware**. New Jersey, US: Prentice Hall, 2006. cap. 10, p. 229–256.

CAPORUSCIO, M.; FUNARO, M.; GHEZZI, C.; ISSARNY, V. ubirest: A restful service-oriented middleware for ubiquitous networking. In: BOUGUETTAYA, A.; SHENG, Q. Z.; DANIEL, F. (Ed.). **Advanced Web Services**. New York, NY, USA: Springer New York, 2014. p. 475–500. ISBN 978-1-4614-7534-7.

CAPORUSCIO, M.; RAVERDY, P.-G.; ISSARNY, V. ubisoap: A service-oriented middleware for ubiquitous networking. **Services Computing, IEEE Transactions on**, IEEE Computer Society, Washington, DC, USA, v. 5, n. 1, p. 86–98, Jan 2012. ISSN 1939-1374.

CARRIERO, N.; GELERNTER, D. Linda in context. **Commun. ACM**, ACM, New York, NY, USA, v. 32, n. 4, p. 444–458, abr. 1989. ISSN 0001-0782.

CASTELLI, G.; MAMEI, M.; ZAMBONELLI, F. The changing role of pervasive middleware: From discovery and orchestration to recommendation and planning. In: **Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on**. Washington, DC, USA: IEEE Computer Society, 2011. p. 214–219.

CHHABRA, J. K.; GUPTA, V. A survey of dynamic software metrics. **Journal of Computer Science and Technology**, Springer US, New York, NY, USA, v. 25, n. 5, p. 1016–1029, 2010. ISSN 1000-9000.

CHOUITEN, M.; DIDIER, J.-Y.; MALLEM, M. Component-based middleware for distributed augmented reality applications. In: **Proceedings of the 5th International Conference on Communication System Software and Middleware**. New York, NY, USA: ACM, 2011. (COMSWARE '11), p. 3:1–3:7. ISBN 978-1-4503-0560-0.

CONTI, M.; DAS, S. K.; BISDIKIAN, C.; KUMAR, M.; NI, L. M.; PASSARELLA, A.; ROUSSOS, G.; TRöSTER, G.; TSUDIK, G.; ZAMBONELLI, F. Fast track article: Looking ahead in pervasive computing: Challenges and opportunities in the era of cyber-physical convergence. **Pervasive Mob. Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 8, n. 1, p. 2–21, fev. 2012. ISSN 1574-1192.

CONTI, M.; GIORDANO, S.; MAY, M.; PASSARELLA, A. From opportunistic networks to opportunistic computing. **Comm. Mag.**, IEEE Press, Piscataway, NJ, USA, v. 48, n. 9, p. 126–139, set. 2010. ISSN 0163-6804.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems: Concepts and Design**. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011, 9780132143011.

CRNKOVIC, I.; SCHMIDT, H. W.; STAFFORD, J. A.; WALLNAU, K. C. 6th icse workshop on component-based software engineering: automated reasoning and prediction. **ACM SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 29, n. 3, p. 1–7, 2004.

CURINO, C.; GIANI, M.; GIORGETTA, M.; GIUSTI, A.; MURPHY, A. L.; PICCO, G. P. Mobile data collection in sensor networks: The tinylime middleware. **Pervasive Mob. Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 1, n. 4, p. 446–469, 2005. ISSN 1574-1192.

DAVID, L.; VASCONCELOS, R.; ALVES, L.; ANDRE, R.; ENDLER, M. A dds-based middleware for scalable tracking, communication and collaboration of mobile nodes. **Journal of Internet Services and Applications**, Springer London, London, UK, v. 4, n. 1, 2013. ISSN 1867-4828.

DE, S.; GOSWAMI, D.; NANDI, S. Consistent coordination decoupling in tuple space based mobile middleware: Design and formal specifications. In: HOTA, C.; SRIMANI, P. (Ed.). **Distributed Computing and Internet Technology**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7753). p. 220–231. ISBN 978-3-642-36070-1.

DIMITRAKOPOULOS, G.; DEMESTICHAS, P. Intelligent transportation systems. **Vehicular Technology Magazine, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 5, n. 1, p. 77–84, March 2010. ISSN 1556-6072.

DOBSON, S.; DENAZIS, S.; FERNáNDEZ, A.; GAÏTI, D.; GELENBE, E.; MASSACCI, F.; NIXON, P.; SAFFRE, F.; SCHMIDT, N.; ZAMBONELLI, F. A survey of autonomic communications. **ACM Trans. Auton. Adapt. Syst.**, ACM, New York, NY, USA, v. 1, n. 2, p. 223–259, dez. 2006. ISSN 1556-4665.

DVINSKY, A.; FRIEDMAN, R. Chameleon – a group communication framework for smartphones. **Software: Practice and Experience**, John & Sons, Ltd, New York, NY, USA, 2014. ISSN 1097-024X.

ESCOFFIER, C.; HALL, R.; LALANDA, P. ipojo: an extensible service-oriented component framework. In: **Services Computing, 2007. SCC 2007. IEEE International Conference on**. Washington, DC, USA: IEEE Computer Society, 2007. p. 474–481.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, jun. 2003. ISSN 0360-0300.

EVERS, C.; KNIEWEL, R.; GEIHS, K.; SCHMIDT, L. The user in the loop: Enabling user participation for self-adaptive applications. **Future Gener. Comput. Syst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 34, p. 110–123, maio 2014. ISSN 0167-739X.

FAISON, T. **Event-Based Programming: Taking Events to the Limit**. 1st. ed. Berkely, CA, USA: Apress, 2011. ISBN 1430243260, 9781430243267.

FLOCH, J.; FRA, C.; FRICKE, R.; GEIHS, K.; WAGNER, M.; LORENZO, J.; SOLADANA, E.; MEHLHASE, S.; PASPALLIS, N.; RAHNAMA, H.; RUIZ, P.; SCHOLZ, U. Playing music - building context-aware and self-adaptive mobile applications. **Software: Practice and Experience**, John Wiley Sons, Ltd, New York, NY, USA, v. 43, n. 3, p. 359–388, 2013. ISSN 1097-024X.

FONTELES, A. S.; NETO, B. J. A.; MAIA, M.; VIANA, W.; ANDRADE, R. M. C. An adaptive context acquisition framework to support mobile spatial and context-aware applications. In: **Proceedings of the 12th International Conference on Web and Wireless Geographical Information Systems**. Berlin, Heidelberg: Springer-Verlag, 2013. (W2GIS'13), p. 100–116. ISBN 978-3-642-37086-1.

GARLAN, D.; MONROE, R. T.; WILE, D. Foundations of component-based systems. In: LEAVENS, G. T.; SITARAMAN, M. (Ed.). New York, NY, USA: Cambridge University Press, 2000. cap. Acme: Architectural Description of Component-based Systems, p. 47–67. ISBN 0-521-77164-1.

GARLAN, D.; SCHMERL, B. Model-based adaptation for self-healing systems. In: **Proceedings of the First Workshop on Self-healing Systems**. New York, NY, USA: ACM, 2002. (WOSS '02), p. 27–32. ISBN 1-58113-609-9.

GRUN, C.; WERTHNER, H.; PRöLL, B.; RETSCHITZEGGER, W.; SCHWINGER, W. Assisting tourists on the move- an evaluation of mobile tourist guides. In: **Proceedings of the 2008 7th International Conference on Mobile Business**. Washington, DC, USA: IEEE Computer Society, 2008. (ICMB '08), p. 171–180. ISBN 978-0-7695-3260-8.

GUI, N.; FLORIO, V. D.; SUN, H.; BLONDIA, C. Toward architecture-based context-aware deployment and adaptation. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 84, n. 2, p. 185–197, fev. 2011. ISSN 0164-1212.

GUINARD, D.; TRIFA, V.; KARNOUSKOS, S.; SPIESS, P.; SAVIO, D. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. **Services Computing, IEEE Transactions on**, IEEE Computer Society, Washington, DC, USA, v. 3, n. 3, p. 223–235, July 2010. ISSN 1939-1374.

HANDTE, M.; SCHIELE, G.; MATJUNTKE, V.; BECKER, C.; MARRóN, P. J. 3pc: System support for adaptive peer-to-peer pervasive computing. **ACM Transactions on Autonomous and Adaptive Systems**, ACM, New York, NY, USA, v. 7, n. 1, p. 1–19, apr 2012. ISSN 15564665.

HELAL, S.; MANN, W.; EL-ZABADANI, H.; KING, J.; KADDOURA, Y.; JANSEN, E. The gator tech smart house: A programmable pervasive space. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 38, n. 3, p. 50–60, mar. 2005. ISSN 0018-9162.

HINZE, A.; SACHS, K.; BUCHMANN, A. Event-based applications and enabling technologies. In: **Proceedings of the Third ACM International Conference on Distributed Event-Based Systems**. New York, NY, USA: ACM, 2009. (DEBS '09), p. 1:1–1:15. ISBN 978-1-60558-665-6.

HOANG, D. D.; PAIK, H.; KIM, C. Service-oriented middleware architectures for cyber-physical systems. **International Journal of Computer Science and Network Security**, s.n., s.l., v. 12, n. 1, p. 79–87, 2012.

HOLLER, J.; TSIATSIS, V.; MULLIGAN, C.; AVESAND, S.; KARNOUSKOS, S.; BOYLE, D. **From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence**. 1st. ed. Orlando, FL, USA: Academic Press, Inc, 2014. ISBN 012407684X, 9780124076846.

HUANG, C.-M.; LAN, K.-c.; TSAI, C.-Z. A survey of opportunistic networks. In: **Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops**. Washington, DC, USA: IEEE Computer Society, 2008. (AINAW '08), p. 1672–1677. ISBN 978-0-7695-3096-3.

HUANG, Y.; YU, J.; CAO, J.; MA, X.; TAO, X.; LU, J. Checking behavioral consistency constraints for pervasive context in asynchronous environments. **CoRR**, s.n., s.l., abs/0911.0136, 2009.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing—degrees, models, and applications. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 40, n. 3, p. 7:1–7:28, ago. 2008. ISSN 0360-0300.

HUGHES, D.; THOELEN, K.; HORRé, W. LooCI: a loosely-coupled component infrastructure for networked embedded systems. **Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia**, ACM, New York, NY, USA, p. 195–203, 2009.

IBM Corp. **An architectural blueprint for autonomic computing**. USA: IBM Corp., 2004.

ISSARNY, V.; GEORGANTAS, N.; HACHEM, S.; ZARRAS, A.; VASSILIADIST, P.; AUTILI, M.; GEROSA, M. A.; HAMIDA, A. Service-oriented middleware for the future internet: state of the art and research directions. **Journal of Internet Services and Applications**, Springer-Verlag, Berlin, Heidelberg, v. 2, n. 1, p. 23–45, 2011. ISSN 1867-4828.

JIAO, W.; SUN, Y. Supporting adaptation of decentralized software based on application scenarios. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 7, p. 1891 – 1906, 2013. ISSN 0164-1212.

KAO, H.-A.; JIN, W.; SIEGEL, D.; LEE, J. A cyber physical interface for automation systems—methodology and examples. **Machines**, s.n., s.l., v. 3, n. 2, p. 93, 2015. ISSN 2075-1702.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 1, p. 41–50, jan. 2003. ISSN 0018-9162.

KRAMER, D.; KOCUROVA, A.; OUSSENA, S.; CLARK, T.; KOMISARCZUK, P. An extensible, self contained, layered approach to context acquisition. In: **Proceedings of the Third International Workshop on Middleware for Pervasive Mobile and Embedded Computing**. New York, NY, USA: ACM, 2011. (M-MPAC '11), p. 6:1–6:7. ISBN 978-1-4503-1065-9.

KRAMER, J.; MAGEE, J. Self-managed systems: an architectural challenge. In: **2007 Future of Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 259–268. ISBN 0-7695-2829-5.

KRUPITZER, C.; ROTH, F. M.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C. A survey on engineering approaches for self-adaptive systems. **Pervasive and Mobile Computing**, IEEE Computer Society, Washington, DC, USA, n. 0, p. –, 2014. ISSN 1574-1192.

LAU, K.-K.; WANG, Z. Software component models. **Software Engineering, IEEE Transactions on**, IEEE Computer Society, Washington, DC, USA, v. 33, n. 10, p. 709–724, Oct 2007. ISSN 0098-5589.

LEE, C.; LEE, G. M.; RHEE, W. S. Standardization and challenges of smart ubiquitous networks in itu-t. **Communications Magazine, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 51, n. 10, p. 102–110, October 2013. ISSN 0163-6804.

LEE, E. A. The past, present and future of cyber-physical systems: A focus on models. **Sensors**, s.n., s.l., n. 3, p. 4837, 2015. ISSN 1424-8220.

LEMOS, R. de; GIESE, H.; MÜLLER, H. A.; SHAW, M.; ANDERSSON, J.; BARESI, L.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CUKIC, B.; DESMARAIS, R.; DUSTDAR, S.; ENGELS, G.; GEIHS, K.; GOESCHKA, K. M.; GORLA, A.; GRASSI, V.; INVERARDI, P.; KARSAI, G.; KRAMER, J.; LITOIU, M.; LOPES, A.; MAGEE, J.; MALEK, S.; MANKOVSKII, S.; MIRANDOLA, R.; MYLOPOULOS, J.; NIERSTRASZ, O.; PEZZÈ, M.; PREHOFER, C.; SCHÄFER, W.; SCHLICHTING, R.; SCHMERL, B.; SMITH, D. B.; SOUSA, J. P.; TAMURA, G.; TAHVILDARI, L.; VILLEGAS, N. M.; VOGEL, T.; WEYNS, D.; WONG, K.; WUTTKE, J. Software engineering for self-adaptive systems: A second research roadmap. In: LEMOS, R. de; GIESE, H.; MÜLLER, H. A.; SHAW, M. (Ed.). **Software Engineering for Self-Adaptive Systems II**. [S.l.]: Springer-Verlag, 2013. v. 7475, p. 1–32. ISBN 978-3-642-35813-5.

LI, Q.; LI, H.; RUSSELL, P.; CHEN, Z.; WANG, C. Ca-p2p: context-aware proximity-based peer-to-peer wireless communications. **Communications Magazine, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 52, n. 6, p. 32–41, June 2014. ISSN 0163-6804.

LIMA, F. F.; ROCHA, L. S.; MAIA, P. H.; ANDRADE, R. M. A decoupled and interoperable architecture for coordination in ubiquitous systems. **2013 VII Brazilian Symposium on Software Components, Architectures and Reuse**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 31–40, 2011.

LIMA, F. F. P.; ROCHA, L. S.; MAIA, P. H. M.; ANDRADE, R. M. C. A decoupled and interoperable architecture for coordination in ubiquitous systems. In: **Proceedings of the 2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse**. Washington, DC, USA: IEEE Computer Society, 2011. (SBCARS '11), p. 31–40. ISBN 978-0-7695-4626-1.

LIN, S.; TAÏANI, F.; BERTIER, M.; BLAIR, G.; KERMARREC, A.-M. Transparent componentisation: High-level (re)configurable programming for evolving distributed systems. In: **Proceedings of the 2011 ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2011. (SAC '11), p. 203–208. ISBN 978-1-4503-0113-8.

LORENZ, A. Architectural patterns for applications with external user interface elements. **Pervasive Mob. Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 9, n. 2, p. 269–280, abr. 2013. ISSN 1574-1192.

LU, H.; ZHOU, Y.; XU, B.; LEUNG, H.; CHEN, L. The ability of object-oriented metrics to predict change-proneness: A meta-analysis. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 17, n. 3, p. 200–242, jun. 2012. ISSN 1382-3256.

MAIA, M.; ROCHA, L.; MAIA, P.; ANDRADE, R. An autonomous middleware model for essential services in distributed mobile applications. In: VENKATASUBRAMANIAN, N.; GETOV, V.; STEGLICH, S. (Ed.). **Mobile Wireless Middleware, Operating Systems, and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, v. 93). p. 57–70. ISBN 978-3-642-30606-8.

MAIA, M. E.; ROCHA, L. S.; ANDRADE, R. M. Requirements and challenges for building service-oriented pervasive middleware. In: **Proceedings of the 2009 International Conference on Pervasive Services**. New York, NY, USA: ACM, 2009. (ICPS '09), p. 93–102. ISBN 978-1-60558-644-1.

MAIA, M. E. F.; ANDRADE, R. M. C.; FILHO, C. A. B. d. Q.; BRAGA, R. B.; AGUIAR, S.; MATEUS, B. G.; NOGUEIRA, R.; TOORN, F. Usable – a communication framework for ubiquitous systems. In: **Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications**. Washington, DC, USA: IEEE Computer Society, 2014. (AINA '14), p. 81–88. ISBN 978-1-4799-3630-4.

MAIA, M. E. F.; FILHO, J. B. F.; FILHO, C. A. B. de Q.; CASTRO, R. N. S.; ANDRADE, R. M. C.; TOORN, F. Framework for building intelligent mobile social applications. In: **Proceedings of the 27th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2012. (SAC '12), p. 525–530. ISBN 978-1-4503-0857-1.

MAIA, M. E. F.; FONTELES, A.; NETO, B.; GADELHA, R.; VIANA, W.; ANDRADE, R. M. C. Loccam - loosely coupled context acquisition middleware. In: **Proceedings of the 28th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2013. (SAC '13), p. 534–541. ISBN 978-1-4503-1656-9.

MAITLAND, J.; MCGEE-LENNON, M.; MULVENNA, M. Pervasive healthcare: from orange alerts to mindcare. **SIGHIT Rec.**, ACM, New York, NY, USA, v. 1, n. 1, p. 38–40, mar. 2011. ISSN 2158-8813.

MAKRIS, P.; SKOUTAS, D.; SKIANIS, C. A survey on context-aware mobile and wireless networking: On networking and computing environments' integration. **Communications Surveys Tutorials, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 15, n. 1, p. 362–386, 2013. ISSN 1553-877X.

MAMEI, M.; ZAMBONELLI, F. **Field-Based Coordination for Pervasive Multiagent Systems (Springer Series on Agent Technology)**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540279687.

MAMEI, M.; ZAMBONELLI, F. Programming pervasive and mobile computing applications. **ACM Transactions on Software Engineering and Methodology**, ACM, New York, NY, USA, v. 18, n. 4, p. 1–56, jul. 2009. ISSN 1049331X.

MAMEI, M.; ZAMBONELLI, F. Programming pervasive and mobile computing applications: The tota approach. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, New York, NY, USA, p. 1–53, 2009.

MAUREL, Y.; LALANDA, P.; DIACONESCU, A. Towards introspectable, adaptable and extensible autonomic managers. In: **Proceedings of the 7th International Conference on Network and Services Management**. Laxenburg, Austria, Austria: International Federation for Information Processing, 2011. (CNSM '11), p. 479–483. ISBN 978-3-901882-44-9.

MAYER, S.; TSCHOFEN, A.; DEY, A. K.; MATTERN, F. User interfaces for smart things – a generative approach with semantic interaction descriptions. **ACM Trans. Comput.-Hum. Interact.**, ACM, New York, NY, USA, v. 21, n. 2, p. 12:1–12:25, fev. 2014. ISSN 1073-0516.

MEDVIDOVIC, N.; EDWARDS, G. Software architecture and mobility: A roadmap. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 83, n. 6, p. 885–898, jun. 2010. ISSN 0164-1212.

MOBILINE: A Nested Software Product Line for the domain of mobile and context-aware applications. **Science of Computer Programming**, v. 78, n. 12, p. 2381–2398, 2013. ISSN 0167-6423. Special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011).

MORIN, B.; BARAIS, O.; JEZEQUEL, J.; FLEUREY, F.; SOLBERG, A. Models@ run.time to support dynamic adaptation. **Computer**, v. 42, n. 10, p. 44–51, Oct 2009. ISSN 0018-9162.

MUHL, G.; PARZYJEGLA, H.; PRELLWITZ, M. Analyzing content-based publish/subscribe systems. In: **Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems**. New York, NY, USA: ACM, 2015. (DEBS '15), p. 128–139. ISBN 978-1-4503-3286-6.

MURPHY, A.; PICCO, G. LIME: A coordination model and middleware supporting mobility of hosts and agents. **ACM Transactions on Software**, IEEE Computer Society, Washington, DC, USA, v. 15, n. 3, p. 279–328, 2006.

NETO, B. J.; ANDRADE, R.; MAIA, M. E.; FONTELES, A.; VIANA, W. A coordination framework for dynamic adaptation in ubiquitous systems based on distributed tuple space. In: **Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International**. Washington, DC, USA: IEEE Computer Society, 2013. p. 1430–1435.

NETO, B. J. A.; ANDRADE, R. M. C.; MAIA, M. E. F.; FONTELES, A. S.; VIANA, W. A coordination framework for dynamic adaptation in ubiquitous systems based on distributed tuple space. In: **International Conference on Wireless Communications and Mobile Computing - IWCMC**. Washington, DC, USA: IEEE Computer Society, 2013. p. 1430–1435.

OFFUTT, J.; ABDURAZIK, A.; SCHACH, S. R. Quantitatively measuring object-oriented couplings. **Software Quality Control**, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, n. 4, p. 489–512, dez. 2008. ISSN 0963-9314.

PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. Context aware computing for the internet of things: A survey. **Communications Surveys Tutorials, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 16, n. 1, p. 414–454, First 2014. ISSN 1553-877X.

PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. Context aware computing for the internet of things: A survey. **Communications Surveys Tutorials, IEEE**, IEEE Computer Society, Washington, DC, USA, v. 16, n. 1, p. 414–454, First 2014. ISSN 1553-877X.

PERKINS, C. E.; ROYER, E. M. Ad-hoc on-demand distance vector routing. In: **In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications**. Washington, DC, USA: IEEE Computer Society, 1997. p. 90–100.

PREUVENEERS, D.; BERBERS, Y. Towards context-aware and resource-driven self-adaptation for mobile handheld applications. In: **Proceedings of the 2007 ACM symposium on Applied computing**. New York, NY, USA: ACM, 2007. (SAC '07), p. 1165–1170. ISBN 1-59593-480-4.

RAMBOLD, M.; KASINGER, H.; LAUTENBACHER, F.; BAUER, B. Towards autonomic service discovery a survey and comparison. In: **Services Computing, 2009. SCC '09. IEEE International Conference on**. Washington, DC, USA: IEEE Computer Society, 2009. p. 192–201.

RODRÍGUEZ, M. D.; GONZALEZ, V. M.; FAVELA, J.; SANTANA, P. C. Home-based communication system for older adults and their remote family. **Comput. Hum. Behav.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 25, n. 3, p. 609–618, maio 2009. ISSN 0747-5632.

SANTANNA, C.; FIGUEIREDO, E.; GARCIA, A.; LUCENA, C. J. P. On the modularity of software architectures: A concern-driven measurement framework. In: **Proceedings of the First European Conference on Software Architecture**. Berlin, Heidelberg: Springer-Verlag, 2007. (ECSA'07), p. 207–224. ISBN 3-540-75131-9, 978-3-540-75131-1.

SARKAR, N. I.; LOL, W. G. A study of manet routing protocols: Joint node density, packet length and mobility. In: **Computers and Communications (ISCC), 2010 IEEE Symposium on**. Washington, DC, USA: IEEE Computer Society, 2010. p. 515 –520. ISSN 1530-1346.

SASSI, A.; MAMEI, M.; ZAMBONELLI, F. Towards a general infrastructure for location-based smart mobility services. In: **High Performance Computing Simulation (HPCS), 2014 International Conference on**. Washington, DC, USA: IEEE Computer Society, 2014. p. 849–856.

SCHIELE, G. **System Support for Spontaneous Pervasive Computing Environments**. Tese (Doutorado) — University of Stuttgart - Department of Computer Science, 2007.

SCHUSTER, D.; ROSI, A.; MAMEI, M.; SPRINGER, T.; ENDLER, M.; ZAMBONELLI, F. Pervasive social context: Taxonomy and survey. **ACM Trans. Intell. Syst. Technol.**, ACM, New York, NY, USA, v. 4, n. 3, p. 46:1–46:22, jul. 2013. ISSN 2157-6904.

SEINTURIER, L.; MERLE, P.; ROUVOY, R.; ROMERO, D.; SCHIAVONI, V.; STEFANI, J.-B. A component-based middleware platform for reconfigurable service-oriented architectures. **Software: Practice and Experience**, John Wiley & Sons, Ltd, New York, NY, USA, v. 42, n. 5, p. 559–583, 2012. ISSN 1097-024X.

SONG, J.; KUNZ, A.; SCHMIDT, M.; SZCZYTOWSKI, P. Connecting and Managing M2M Devices in the Future Internet. **Mobile Networks and Applications**, Springer Berlin Heidelberg, Berlin, Heidelberg, v. 19, n. 1, p. 4–17, 2014. ISSN 1383-469X.

SOUZA, R. S. D.; LOPES, J. a. L.; GADOTTI, G. I.; aO, M. Z. G.; YAMIN, A.; GEYER, C. A Scalable and Proactive Coordination Model for Ubiquitous Applications. In: **IV Brazilian Symposium on Pervasive and Ubiquitous Computing - SBCUP**. Curitiba: s.n., 2012. (in Portuguese).

STEVENS, W.; MYERS, G.; CONSTANTINE, L. Classics in software engineering. In: YOURDON, E. N. (Ed.). Upper Saddle River, NJ, USA: Yourdon Press, 1979. cap. Structured Design, p. 205–232. ISBN 0-917072-14-6.

SUZUKI, T.; PINTE, K.; CUTSEM, T. V.; MEUTER, W. D.; YONEZAWA, A. Programming language support for routing in pervasive networks. In: **Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on**. [S.l.: s.n.], 2011. p. 226–232.

THOELEN, K.; PREUVENEERS, D.; MICHIELS, S.; JOOSEN, W.; HUGHES, D. Types in their prime: Sub-typing of data in resource constrained environments. In: STOJMENOVIC, I.; CHENG, Z.; GUO, S. (Ed.). **Mobile and Ubiquitous Systems: Computing, Networking, and Services**. Berlin, Heidelberg: Springer International Publishing, 2014, (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, v. 131). p. 250–261. ISBN 978-3-319-11568-9.

THOMA, M.; BRAUN, T.; MAGERKURTH, C.; ANTONESCU, A.-F. Managing things and services with semantics: A survey. In: **Network Operations and Management Symposium (NOMS), 2014 IEEE**. Washington, DC, USA: IEEE Computer Society, 2014. p. 1–5.

TRILLES, S.; LUJÁN, A.; BELMONTE, Ó.; MONTOLIU, R.; TORRES-SOSPEDRA, J.; HUERTA, J. Senviro: A sensorized platform proposal using open hardware and open standards.

Sensors, IEEE Computer Society, Washington, DC, USA, v. 15, n. 3, p. 5555, 2015. ISSN 1424-8220.

VISWANATH, K.; OBRACZKA, K. Modeling the performance of flooding in wireless multi-hop ad hoc networks. **Comput. Commun.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 29, n. 8, p. 949–956, maio 2006. ISSN 0140-3664.

VLIST, B.; NIEZEN, G.; RAPP, S.; HU, J.; FEIJS, L. Configuring and controlling ubiquitous computing infrastructure with semantic connections: A tangible and an ar approach. **Personal Ubiquitous Comput.**, Springer-Verlag, London, UK, UK, v. 17, n. 4, p. 783–799, abr. 2013. ISSN 1617-4909.

WEI, E. J.; CHAN, A. T. CAMPUS: A middleware for automated context-aware adaptation decision making at run time. **Pervasive and Mobile Computing**, Elsevier B.V., New York, NY, USA, v. 9, n. 1, p. 35–56, fev. 2013. ISSN 15741192.

WEISER, M. The computer for the 21st century. **SIGMOBILE Mob. Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 3, n. 3, p. 3–11, jul. 1999. ISSN 1559-1662.

WEYNS, D.; IFTIKHAR, M. U.; SöDERLUND, J. Do external feedback loops improve the design of self-adaptive systems? a controlled experiment. In: **Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**. Piscataway, NJ, USA: IEEE Press, 2013. (SEAMS '13), p. 3–12. ISBN 978-1-4673-4401-2.

WEYNS, D.; MALEK, S.; ANDERSSON, J. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In: **Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems**. New York, NY, USA: ACM, 2010. (SEAMS '10), p. 84–93. ISBN 978-1-60558-971-8.

WEYNS, D.; SCHMERL, B.; GRASSI, V.; MALEK, S.; MIRANDOLA, R.; PREHOFER, C.; WUTTKE, J.; ANDERSSON, J.; GIESE, H.; GOSCHKA, K. On patterns for decentralized control in self-adaptive systems. In: LEMOS, R. de; GIESE, H.; MULLER, H.; SHAW, M. (Ed.). **Software Engineering for Self-Adaptive Systems II**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7475). p. 76–107. ISBN 978-3-642-35812-8.

WHITBECK, J.; CONAN, V. Hymad: Hybrid dtn-manet routing for dense and highly dynamic wireless networks. **Comput. Commun.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 33, n. 13, p. 1483–1492, ago. 2010. ISSN 0140-3664.

WISSEN, B. van; PALMER, N.; KEMP, R.; KIELMANN, T.; BAL, H. ContextDroid: an expression-based context framework for Android. In: **Proceedings of PhoneSense 2010**. s.l.: s.n., 2010.

WOLF, T. D.; HOLVOET, T. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. In: **Industrial Informatics, 2003. INDIN 2003. Proceedings. IEEE International Conference on**. Washington, DC, USA: IEEE Computer Society, 2003. p. 470–479.

WOLF, T. D.; HOLVOET, T. Engineering self-organising systems. In: BRUECKNER, S. A.; SERUGENDO, G. M.; KARAGEORGOS, A.; NAGPAL, R. (Ed.). Berlin, Heidelberg:

Springer-Verlag, 2005. cap. Emergence Versus Self-organisation: Different Concepts but Promising when Combined, p. 1–15. ISBN 3-540-26180-X, 978-3-540-26180-3.

ZAMBONELLI, F.; VIROLI, M. A survey on nature-inspired metaphors for pervasive service ecosystems. **International Journal of Pervasive Computing and Communications**, Emerald Publishing Limited, s.l., v. 7, n. 3, p. 186–204, 2011.

ZHANG, W.; HANSEN, K. M.; INGSTRUP, M. A hybrid approach to self-management in a pervasive service middleware. **Know.-Based Syst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 67, p. 143–161, set. 2014. ISSN 0950-7051.

ZHANG, Y.; DUAN, L.; CHEN, J. L. Event-driven soa for iot services. In: **Services Computing (SCC), 2014 IEEE International Conference on**. Washington, DC, USA: IEEE Computer Society, 2014. p. 629–636.

ZHAO, Y.; MA, D.; HU, C.; LIU, M.; HUANG, Y. Socom: A service-oriented collaboration middleware for multi-user interaction with web services based scientific resources. In: **Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on**. Washington, DC, USA: IEEE Computer Society, 2007. p. 28–28.

ZHOU, Y.; MA, X.; GALL, H. A middleware platform for the dynamic evolution of distributed component-based systems. **Computing**, Springer, Berlin, Heidelberg, v. 96, n. 8, p. 725–747, 2014. ISSN 0010-485X.

APPENDIX A - COUPLING IN CYBER-PHYSICAL SYSTEMS

This appendix discusses the impact coupling has on software systems, particularly focusing on coupling among CPS, along with available mechanisms to measure how coupled one system is.

Introduction

The definition of software coupling dates back to 1979, when Stevens et. al. defined coupling as the association strength between two software modules (STEVENS *et al.*, 1979), indicating how interdependent these modules are. Systems with strong coupling among its modules are more complex do understand, test and maintain.

Hence, coupling has long been used as an indicator of software quality (OFFUTT *et al.*, 2008). To assess how coupled a system is, software engineers analyze its design to understand how its modules are interconnected. Two axioms are important to design uncoupled systems (FAISON, 2011):

- Axiom 1. Complex classes or components should be as uncoupled as possible.
- Axiom 2. Coupling, when inevitable, should be introduced into simpler classes or components first.

As an indication of how coupling affects a system, imagine two modules, where A requires B to be present. The degree with which changes in B affect A is direct related to how coupled these two modules are. If A and B are strongly coupled, changes in B affect A more deeply. Opponsingly, if the interconnection among A and B is weak, changes in B are hardly noticeable by A.

The same principle applies to interacting modules in CPS. Uncoupled systems are less affected by changes, failures or disconnection periods than systems with strong relations among its modules are. Since unpredictability is a strong characteristic in CPS, mechanisms to minimize the complexity to deal with failures are required.

In that direction, even though some level of coupling must always be present, since distributed modules interact via message passing, which by itself introduces coupling, the goal is to use primitives to minimize coupling, but are able to handle unpredicted situations as well. The following sections discuss existing coupling types, metrics, along with mechanisms to minimize coupling.

Coupling types

When designing a system, to tame complexity, software engineers divide it into modules and interconnections. The relationship between system modules can be of two main types (FAISON, 2011): i) static, where interacting modules must be available at compile-time; and ii) dynamic, where the interaction is based on the modules' interfaces, but the executables are only necessary at runtime. It is advisable that complex modules interact based on dynamic coupling.

There are three main types of coupling (FAISON, 2011): i) logic, when two classes make assumptions about each other, even if they do not interact directly; ii) type, when one class or component requires another to be present based on its interface; and iii) signature, which defines coupling based only on the input and output parameters of the required methods.

Logic coupling connects two classes or components based on the algorithms they implement. In some cases, they do not interact directly, but strongly affect and influence one another. For instance, imagine an algorithm to compress a file and another to decompress. These two algorithms are strongly related, and a good design would not split them up into two separate classes or modules.

Type coupling is the most used coupling mechanism in system development. Using type coupling, the system design is based on class and component types (e.g. interfaces or abstract classes in object-oriented design), but the actual implementation is only defined at runtime.

Signature coupling is more subtle and introduces flexibility in the development and execution. It is based on the method signature required by one class or component, defined by the input and output parameters, pre- and post-conditions and invariants. Sometimes it is considered a subclass of the Type coupling, but at a smaller granularity.

In CPS, dynamic coupling should be used whenever possible. The interaction based on types and signature acts as an agreement between distributed system modules on the data format and on the method functionality, permitting the development of each module independently of the rest of the system.

Coupling metrics

Coupling is inevitable in CPS. However, because of the characteristics of these systems (e.g. unpredictability, volatility), its design must minimize the coupling among complex parts of the system. In order to assess how coupled a system is, researchers rely on software coupling metrics (LU *et al.*, 2012; OFFUTT *et al.*, 2008; CHHABRA; GUPTA, 2010).

Coupling metrics can be divided into static and dynamic metrics. Static metrics capture information about the system design and source code, without executing the system. Dynamic metrics analyze that same system considering runtime interactions among modules.

Static coupling metrics

Static coupling analyzes the source code to measure how coupled the system design is, looking at interconnections among modules (classes, components, subsystems). For instance, in object-oriented analysis, interconnections are inheritance and associations among classes.

In this doctoral thesis, the goal is to analyze CPS considering coupling metrics to understand the ease with which the system can be adapted. Hence, three well known coupling metrics were used (BURROWS *et al.*, 2010): i) Concern diffusion over components (CDC), that counts the number of classes and interfaces implementing a given functionality, along with the number of other classes and interfaces accessing the classes and interfaces that implements that functionality; ii) Concern diffusion over operations (CDO), that counts the number of methods implementing a given functionality, along with the number of times other methods access the methods that implement that functionality; iii) Coupling Between Components (CBC), that counts the number of times two components are linked, by accessing their methods.

Using these metrics, CyberSupport was assessed to understand the impact it introduced considering coupling. One important claim made in the hypothesis is that CyberSupport helped to minimize coupling among applications and sensors/actuators, which was later verified and presented in the performance evaluation in section 5.3.1.

Two design decisions present in CyberSupport helped to decrease the overall coupling. The first is the interaction based on shared spaces, where all interactions between application, nearby devices, services and sensors/actuators took place using SysSU-DTS. Now, instead of an application making direct invocation to components providing sensor data, it interacted with the tuple space. That decision was in conformance with Axiom 2, where all

coupling is introduced into one infrastructure responsible solely for the interaction among system modules.

The second design decision is the use of CIB to access sensor/actuator components. The coupling among application and sensor/actuator is inevitable, but it is based on signature coupling, which is the most desired coupling. To change the component being used required only to update the signature of the component required by the application, since the actual component implementation has no references to the application using it. Another advantage of this approach is in accordance with Axiom 1, because all complexity present in the component implementation is hidden away from the application.

Dynamic coupling metrics

A second set of metrics to evaluate coupling considers the system in execution. Here, instead of looking at the design of a system to understand its interconnections, dynamic metrics aim to measure how coupled the system modules are at runtime (CHHABRA; GUPTA, 2010). For instance, they count the number of messages two modules exchange, the number of methods that are invoked or the number of classes one class interact with at runtime.

These metrics are important to understand, out of the modules one module is coupled with at design time, the ones it really is coupled with at runtime. Although dynamic metrics are harder to collect, because the system must be executed, they give more precise information about the system behavior. This is an important contribution to CPS, where runtime interactions are harder to predict. Hence, the less coupled system modules are, more tailored to volatile environments they become.

Unpredictability in CPS makes design-time assumptions harder to guarantee. Hence, to assume system modules will be available during application execution should be avoided. Instead, development mechanisms to permit the interaction when these modules became available is useful to handle the unpredictability of CPS.

Eugster et. al (EUGSTER *et al.*, 2003) claim that the separation between system modules should be in three separate dimensions: i) time, where interacting entities may exchange messages even if they are not online at the same moment; ii) address, where interactions among devices based on physical addresses should be avoided; and iii) synchrony, where sychronous interactions should take place only in limited situations.

Considering these dimensions, three system execution coupling metrics are proposed

by this doctoral thesis: i) Number of Address References (NAR), which counts the number of times two modules interact based on a network/application address; ii) Number of Online Interactions (NOI), which counts the number of times two modules must be online at the same time for an interaction to take place; and iii) Number of Synchronous Interactions, which counts the number of times a synchronous interaction takes place.

These three metrics give an indication of how coupled two modules are at runtime. Higher coupling means less resiliency to failures, since the overall system only behaves normally when interactng modules are available. For instance, by relying on a network address (NAR) or on the availability of a module at a given time (NOI), the system becomes less prone to behave normally when facing disconnections.

CyberSupport removed the necessity to rely on network addresses in two separate moments. First, the interaction between devices takes places by writing and reading in SysSU-DTS tuples with an agreed description. Second, interaction among applications and sensors/actuators take place using the CIB and CACs, where all the communication with sensors/actuators is encapsulated within a CAC.

The online uncoupling is provided by using the tuple space and pub-sub primitives, where applications manifest their interest in receiving sensor information and CACs place sensor data in the tuple space. Hence, consumers and producers of data might interact even if they are not online at the same time.

One problem here is the time tuples remain valid. Every tuple has a time stamp describing when it was generated and applications are left to decide if tuples are still valid. Additionally, tuples read from nearby devices must have been generated within a time frame to be considered valid.

Conclusions

The separation between interacting modules is an important quality attribute in software systems. The more uncoupled systems are, the more easier it is to adapt and modify them. In CPS, uncoupling is even more important to handle unpredictability.

Uncoupling is important in two levels. Static, analyzing the system design to improve the separation among its modules, and dynamic, to lessen the interconnection strength between two modules at runtime. Hence, this doctoral thesis presented mechanisms to improve both static and dynamic coupling in CPS.