ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

TESI DI LAUREA

in
Sistemi Mobili M

Design and Implementation of a Cloud-based Middleware for Persuasive Android Applications

CANDIDATO
Matteo Lelli

RELATORE
Chiar.mo Prof. Ing. Paolo Bellavista

CORRELATORI
Prof. Abdelsalam Helal
Prof. Ing. Antonio Corradi

Anno Accademico 2014/15

Sessione III

Keywords: *Mobile Systems*

Persuasive Computing

Android Middleware

Cloud Integration

Smartwatch

Table of Contents

Table of Figures

Table of Tables

Table of Code Snippets

Introduction

In last decades, computing technologies and products have become pervasive, and they are now essential part of our lives. Every day they influence us explicitly and implicitly, changing our way of living and our behaviors, either intentionally or unintentionally. For instance, since on-demand videos became available for free on Internet, people have modified their behaviors and habits, preferring this new video streaming services than renting DVDs. Moreover, after first mobile applications were published, our lives have changed so radically that nowadays we cannot imagine, for example, driving in an unknown area without using a navigation app, or having a workout without a fitness mobile application supporting us.

Nevertheless, computers weren't initially created to persuade; they were built for handling, calculating, storing, and retrieving data. But, as computers have migrated from research labs into everyday life, they have become more and more persuasive. Nowadays, computers are assuming a variety of roles as persuaders, including roles that traditionally were filled by teachers, coaches, speakers, doctors, or salespeople. This field of research is called persuasive technology or captology, also defined as the study of interactive computing systems designed to change people's attitudes and behaviors.

Despite the increasing success of captology, there seems to be a lack of both theoretical and practical frameworks that help developers to build mobile applications able to effectively persuade users. However, Ph.D. Helal and Ph.D. Lee's research work at the Persuasive Laboratory of the University of Florida tried to fill this gap. Indeed, they proposed a simple but effective persuasive model that can be simply used by engineering and computer scientists. Moreover, Ph.D. Helal and Ph.D. Lee developed an Android middleware called Cicero, and based on their previous model, that can be used by developers that want create persuasive applications in a very intuitive and powerful way. Indeed, developers can simply select users’ activities to sense, concentrating themselves on the business logic of their application, while the persuasive side is achieved by the framework using the theoretical model.

My work that is at the center of this project thesis focuses on analyzing the middleware just described, and then on finding improvements and enhancements to it. The most important ones are a new sensing architecture, a new cloud-based structure, and a new protocol that lets developers create application tailored also for smartwatches. In particular, the first improvement is fundamental in order to sense multiple behaviors at the same time, an essential future for a persuasive middleware that would like to sense complex situations and behaviors. The second enhancement allows developing concurrent applications on several devices, assessing behaviors’ changes as a cloud-based service. The third main improvement is an important enhancement in order to have a more complete and pervasive sensing to better persuade users that own a smartwatch.

This dissertation is divided into four Chapters that, after an initial background analysis useful in order to understand pre-existent middleware, discuss about models I introduced, and about new implementation challenges I dealt with. In particular, First Chapter introduces captology, and it analyses the need of a persuasive middleware into that context. In Second Chapter I present the theoretical model developed at the Persuasive Laboratory of the University of Florida, focusing on describing Cicero, in particular on the aspect that I am going to improve. In Third Chapter I propose several new models and improvements, among which ones I previously described. Finally, in the Fourth Chapter I present implementation challenges I dealt with in order to port new models and improvements into an Android middleware, and how I solved them. Moreover, I show how I arranged a clarifier use case for the new Cicero middleware, and I present experimental results obtained from several performance tests on it.

Introduzione

Negli ultimi decenni, le tecnologie e i prodotti informatici sono diventati pervasivi e sono ora una parte essenziale delle nostre vite. Ogni giorno ci influenzano in maniera più o meno esplicita, cambiando il nostro modo di vivere e i nostri comportamenti più o meno intenzionalmente. Ad esempio, da quando i video su richiesta sono diventati reperibili gratuitamente su Internet, le persone hanno modificato i loro comportamenti ed abitudini, preferendo questi nuovi servizi di streaming video rispetto al classico noleggio di DVD. Inoltre, dopo la pubblicazione delle prime applicazioni per cellulari smartphone, le nostre vite sono cambiate così radicalmente che oggigiorno non possiamo immaginare, ad esempio, di guidare in una zona sconosciuta senza l'utilizzo di una applicazione di navigazione o di allenarci senza il supporto di un'applicazione di fitness.

Tuttavia, i computer non nacquero inizialmente per persuadere: essi furono costruiti per gestire, calcolare, immagazzinare e recuperare dati. Non appena i computer si sono spostati dai laboratori di ricerca alla vita di tutti i giorni, sono però diventati sempre più persuasivi. Oggigiorno, i computer stanno assumendo una varietà sempre maggiore di ruoli come persuasori, inclusi quelli che tradizionalmente erano ricoperti da insegnanti, allenatori, oratori, dottori o venditori. Questa area di ricerca è chiamata *pesuasive technology* o *captology*, anche definita come lo studio dei sistemi informatici interattivi progettati per cambiare le attitudini e le abitudini delle persone.

Nonostante il successo crescente delle tecnologie persuasive, sembra esserci una mancanza di framework sia teorici che pratici, che possano aiutare gli sviluppatori di applicazioni mobili a costruire applicazioni in grado di persuadere effettivamente gli utenti finali. Tuttavia, il lavoro condotto dal Professor Helal e dal Professor Lee al *Persuasive Laboratory* all'interno dell'*University of Florida* tenta di colmare questa lacuna. Infatti, hanno proposto un modello di persuasione semplice ma efficace, il quale può essere usato in maniera intuitiva da ingegneri o specialisti informatici. Inoltre, il Professor Helal e il Professor Lee hanno anche sviluppato Cicero, un middleware per dispositivi Android basato sul loro precedente modello, il quale può essere usato in modo molto semplice e veloce dagli sviluppatori per creare applicazioni persuasive. Infatti, gli sviluppatori possono facilmente selezionare le attività degli utenti da scansionare, concentrandosi sulla logica applicativa, mentre il lato di persuasione è affidato al framework, il quale si basa sul modello teorico.

Il mio lavoro al centro di questa tesi progettuale si concentra sull'analisi del middleware appena descritto e, successivamente, sui miglioramenti e ampliamenti portati ad esso. I più importanti sono una nuova architettura di *sensing*, una nuova struttura basata sul cloud e un nuovo protocollo che permette di creare applicazioni specifiche per smartwatch. In particolare, il primo miglioramento è fondamentale per analizzare più comportamenti contemporaneamente, una caratteristica basilare per un middleware che vuole analizzare situazioni e comportamenti complessi. La seconda modifica permette lo sviluppo di applicazioni concorrenti su più dispositivi, stimando le modifiche dei comportamenti attraverso un servizio (*as-a-service*) offerto sul cloud. Il terzo ampliamento è un'aggiunta fondamentale a Cicero che permette un *sensing* più completo e pervasivo, in modo da persuadere in maniera più efficace gli utenti dotati di smartwatch.

Questa tesi è suddivisa in quattro Capitoli, i quali, dopo una analisi iniziale utile a comprendere il middleware preesistente, trattano dei modelli che ho introdotto e delle sfide implementative che ho dovuto affrontare. In particolare, il Primo Capitolo introduce la *captology* e analizza la necessità di un middleware all'interno di questo contesto. Nel Secondo Capitolo presento il modello teorico sviluppato all'*University of Florida*, concentrandomi sull'analisi di Cicero, in particolare sugli aspetti che andrò a migliorare. Nel Terzo Capitolo propongo alcuni nuovi modelli e miglioramenti, tra cui quelli introdotti precedentemente. Infine, nel Quarto Capitolo presento le sfide implementative che ho affrontato per trasferire i nuovi modelli e miglioramenti all'interno di un middleware Android. Inoltre, espongo anche come ho creato un caso d'uso esemplificativo per il nuovo middleware Cicero e presento i risultati sperimentali ottenuti da diversi test sulle performance del caso d’uso appena citato.

Chapter 1 - Persuasive Computing

Nowadays, persuasion is pervasive and is part of our lives. Every day, we are affected consciously or unconsciously by persuasive messages especially from mass media, and they try to change our behavior. Indeed, these media, including newspapers, radio, and television, besiege us with persuasive messages to purchase product, and influence, or change our beliefs.

In this chapter, I analyze how computing technologies and products (e.g., email, web, smartphones, smartwatches, mobile applications, and others), that have become pervasive and are an essential part of our lives, are able to persuade us. Additionally, I explore some middleware and frameworks born in order to simply the developing of persuasive mobile applications.

1.1. Captology

With the advent of new information technologies, such as computers, smartphones, and more recently smartwatches, alternative ways to persuade are arisen. Since the *Conference on Human Factors in Computing Systems* in the 1997, the study of computers as persuasive technologies was introduced as a new area of academic inquiry. In order to denote this research field, B. J. Fogg, the pioneer and a leading expert in the latter area of research, introduces the term *captology* building it from an acronym for Computers As Persuasive Technologies [1]. To better clarify the concept of *captology*, according to Fogg the study of computers as persuasive technologies includes the design, research, and analysis of interactive computing products (computers, mobile phones, websites, wireless technologies, mobile applications, video games, etc.) created for the purpose of changing people's attitudes or behaviors. To illustrate that concept, in the Figure 1 it is drawn an Euler diagram in with the *captology* area is the intersection between the computing-technology domain and persuasion's one.

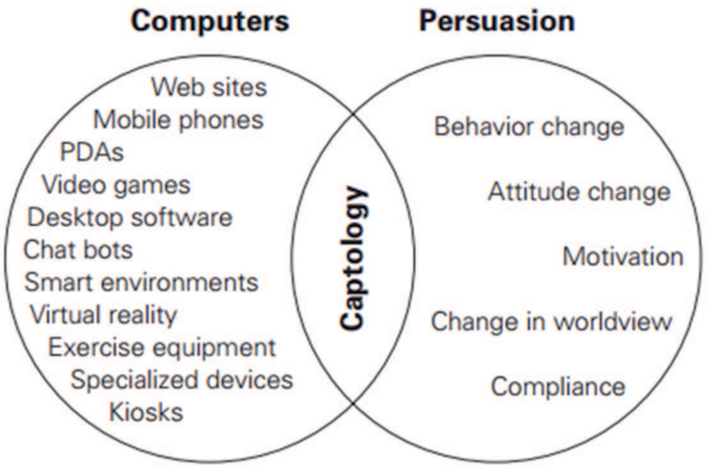


Figure 1: Captology describes the area where computing technology and persuasion overlap

Before going into the details of how computers can be better persuaders than humans, I have to clarify what is a persuasive computer and what is the today's computers function. I will analyze that in the coming sections.

1.1.1. Definition of Persuasive Computers

First of all, the psychology literature suggests many definitions for the word *persuasion*. Fogg synthesizes the various definitions to define *persuasion* as "an attempt to shape, reinforce, or change behaviors, feelings, or thoughts about an issue, object, or action" [1]. In latter definition, it is clear that persuasion implies *intent* to change attitudes or behaviors; in other words, persuasion requires intentionality. Conversely, computers do not have intentions. If so, how can machines persuade us?

Fogg answers to this question proposing that if an intent to change attitudes or behaviors is a factor in the creation, distribution, or adoption of a technology, then that technology inherits a type of intent from human actors. Moreover, Fogg propose three kinds of inherited persuasive intent: endogenous, exogenous, and autogenous.

- Endogenous intent (from within): an endogenous intent comes from those who create or produce the interactive technology. For example, telehealth systems persuade people to develop good health habits.
- Exogenous (caused by external factors): an exogenous intent comes from those who give access to or distribute the interactive technology to others. For instance, a clinician that gives his or her patients a pedometer in hopes that they will become more active.
- Autogenous (self-produced): an autogenous intent comes from the person adopting or using the interactive technology. For example, a person may buy and use a calorie-counting computer device to help change his or her own eating behavior.

Furthermore, it is quite possible that a given interactive technology may fall into more than one category. Despite of this, Fogg find these three categories helpful in better understanding the range and roles of persuasive computing technologies.

1.1.2. A functional view of persuasive computers

Fogg proposes that today's computers function in three basic ways: as tools, as media, and as social actors [1]. As a tool, the computer (or the computer application) allowing people to do things they could not do before, or to do things more easily. Computers also function as media. Indeed, a computer can convey either symbolic content (e.g., text, data graphs) or sensory content (e.g., real-time video, virtual worlds, augmented reality). Computers can also function as social actors. Users seem to respond to computers as social actors when computer technologies adopt animate characteristics (physical features, emotions, voice communication), play animate roles (coach, pet, assistant, opponent), or follow social rules or dynamics (greetings, apologies, turn taking).

Fogg maps these three functions simultaneously into a triangular bi-dimensional space he calls the *Functional Triad*. Figure 2 represents the Functional Triad with some prototypical examples.

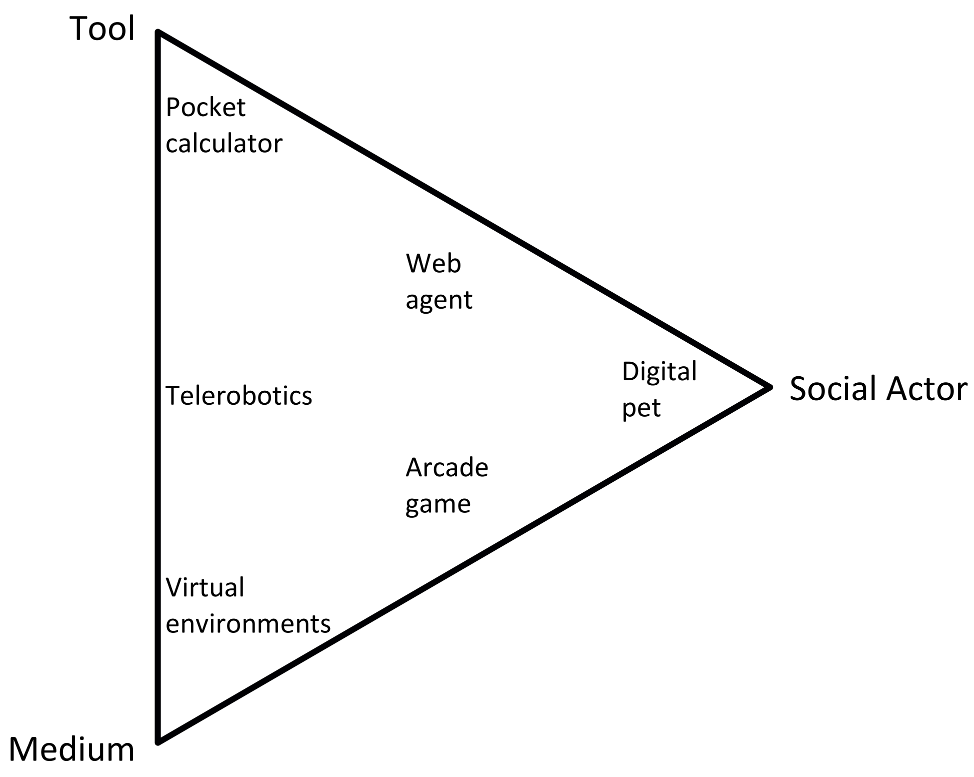


Figure 2: The Function Triad and examples

Moreover, computers functioning as tools, media, or social actors can change attitudes and behaviors through different means. For instance, persuasive technologies can reduce barriers in terms of time, effort, and cost, or they can change mental models and behaviors. Table 1 shows an exhaustive list of examples of how computers can change attitudes and behaviors [1].

Table 1: Three computer functions and their persuasive affordances

Function	Essence	Persuasive affordances
Computer as tool or instrument	Increases capabilities	• Reduces barriers (time, effort, cost) • Increases self-efficiency • Provides information for better decision making • Changes mental models
Computer as medium	Provides experiences	• Provides first-hand learning, insight, visualization, resolve • Promotes understanding of cause/effect relationships • Motivates through experiences, sensation
Computer as social actor	Creates relationships	• Establishes social norms • Invokes social rules and dynamics • Provides social support to sanction

1.1.3. Advantages of persuasive computers

Fogg [2] analyses some advantages of computers as persuader. He finds advantages both over traditional media and over human beings, but he also outlines some pitfalls especially regarding ethical. In fact, persuasive technology can be used in unethical ways in an attempt to change people's attitudes and behaviors. For example, an online game could be used to persuade children to give up personal information.

1.1.3.1. Advantage over Traditional Media

Traditional media, from bumper stickers to radio spots, from print ads to television commercials, have long been used to influence people to change their attitudes or behaviors. But they significantly differ from persuasive technologies for a fundamental feature: *interactivity*.

As a rule of thumb, persuasion techniques are most effective when they are interactive, when persuaders adjust their influence tactics as the situation evolves. For instance, skilled salespeople know this and adjust their pitches according to feedback from the prospect.

Persuasive technologies can adjust what they do based on user inputs, needs, and situations. For example, try to imagine a hypothetical app that helps smokers to quit smoke: this application could provide the right kind of encouragement to help the person quit according to actual progress, adjusting and tailoring supporting messages or prizes. Traditional media aren't able to support this kind of interactivity.

1.1.3.2. Advantages over Humans

Humans are considered good persuaders, but according to Fogg [2] they lose the comparison with persuasive computers and technologies. Indeed, latters have six distinct advantages over human persuaders.

In the first place, they can be **more persistent** than human beings. Indeed, no human can be as persistent as a machine. Computers don't get tired, discouraged, or frustrated. They don't need to eat or sleep. They can work around the clock in active efforts to persuade, or watch and wait for the right moment to intervene.

Second, they **offer greater anonymity**. The option of remaining anonymous is important in sensitive areas such as sexual behavior, substance abuse, or psychological problems. It's often easier (and less embarrassing) to get information or help anonymously, via an interactive computing program, than it is to face another human being. Anonymity also is important when people are experimenting with new attitudes and behaviors. Indeed, anonymity helps overcome social forces that lock people into routines, making it easier for people to change.

Thirdly, computers can store, access, and manipulate **huge volumes of data**, far beyond the capabilities of human beings. This gives interactive technologies the potential to be more persuasive than human beings. For example, persuasive computers are able to use collaborative filtering or automated methods for making inferences in order to predict what a user is likely to buy or do and make recommendations to the user based on that.

In addition, computers can use **many modalities** to influence. Indeed, in order to persuade, computers can present data and graphics, rich audio and video, animation, simulation, hyperlinked content, or a combination of these methods. This skill is very important. In fact, often people are influenced not by information itself but by the modality with which it's presented.

Furthermore, persuasive technologies can **scale easily**, grow quickly when demand increases. Therefore, when persuasion comes to software-based experiences, the ability to scale is relatively easy. You can replicate and distribute persuasive technology experiences that work just like the original. On the other hand, a good human persuader is hard to replicate: if you try, for instance, to increase the person's scope of influence through print, audio, or video communications, the original experience may get lost along the way, particularly if the original experience was interactive.

Finally, computers go where humans cannot go or may not be welcome: they can be **ubiquitous**. It's clear that when interactive computing systems are embedded in everyday objects and environments, they can intervene at precisely the right time and place, giving them greater persuasive power. Furthermore, computing applications are becoming commonplace in locations where human persuaders would not be welcomed, such as the bathroom or bedroom, or where humans cannot go (inside clothing, embedded in an automotive system, or implanted in a toothbrush).

1.1.4. Application domains

Therefore, it is clear that captology could be very powerful, both over traditional media and over humans. Moreover, persuasive computing's field of research is growing more and more, involving several domains, also quite different from each other. Fogg identifies [2] [3] at least twelve domains in which persuasive computing is (or is going to be) fundamental.

The most obvious domain is in promoting commerce, buying and branding, especially via the Web. While promoting commerce is perhaps the most obvious and lucrative application, at least 11 other domains are potential areas for persuasive technology products. The various domains, along with a sample target behavior change, are summarized in Table 2. The domains in the table reflect how much persuasion is part of ordinary human experience, from personal relationships to environmental conservation. Interactive technologies have been (and will continue to be) created to influence people in these 12 domains, as well as in others that are less apparent.

One of the main macro domain is the healthcare. Indeed, advances in healthcare have led to longer life expectancy so that the elderly population is increasing rapidly. Moreover, about 80% of the elderly age with at least one chronic condition and 50% age with at least two [4]. Industry and academia put forth considerable effort to support independent living as well as to provide cost-effective solutions for successful healthy aging. In this context, a new industry had a great enthusiasm in last years: the telehealth systems, a cost-effective approach that could support independent living. These systems are able not only to sensing, monitoring, and tele-care patients, but they are able to enable effective intervention ad persuasion as well. Therefore, captology is becoming essential in these systems: more the persuasion is strong, more they are efficient. For instance, a telehealth system that helps people quit smoking is more effective if it is able not only to track improvements of users, but it is also able to persuade patients about the harmfulness of smoking.

Besides, as described in following chapters, the work of this thesis is aimed to address precisely the rising domain of the telehealth.

Table 2: Persuasive Technology: Domains and Applications

Domain	Example application	Persuades users to
Commerce	Amazon.com's recommendation system	Buy more books and other products
Education, learning, and training	CodeWarriorU	Engage in activities that promote learning how to write code
Safety	Drunk driving simulator	Avoid driving under the influence of alcohol
Environmental preservation	Scorecard.org	Take action against organizations that pollute
Occupational effectiveness	"In my steps" VR system	Treat cancer patients with more empathy
Preventive healthcare	Quitnet.com	Quit smoking
Fitness	Tectrix VR bike	Exercise and enjoy it
Disease management	Bronkie the bronchiasaurus game	Manage asthma more effectively
Personal finance	FinancialEngines.com	Create and adhere to a retirement plan
Community involvement/activism	CapitolAdvantage.com	Get ordinary citizens involved in public affairs
Personal relationships	Classmates.com	Reconnect with former classmates
Personal management and self-improvement	MyGoals.com	Set goals and take the needed steps to achieve them

1.2. *Need for a persuasive theory*

As described in previous subsections, Fogg introduced a new field of research, and it is becoming more and more notable due to the advent of new pervasive technologies and because of its potential. In this wide open scenario, a lot of researchers tried to use the power of captology, adapting it to many areas. Certainly, two of the most notable fields are the healthcare and the telehealth. In this context, simple and naive approaches for intervention on the behavior alteration do not seem to work. For instance, just telling what people need to do for loose weight or manage some diseases such as diabetes. Consequently, exploiting the persuasive power to make this behavior alteration more effective is a great improvement, mainly for the benefit of patients.

In addition, existing behavior change theories and models explain how people change their behaviors. Each theory and model focuses on different factors such as self-efficacy, intention, social, environmental and personal factors. However, these models are not easily utilizable as is by health Telematics researchers especially engineering and computer scientists.

Therefore, it is clear the need for a behavior change model, more acceptable and utilizable by the health Telematics researcher community. In order to answer to this need, Professor Helal and Doctor Lee developed at the University of Florida (USA) a new behavior model based on actions, called Action-Behavior Model (ABM). I will analyze this model into detail in Chapter 3.

1.3. Need for a middleware

Furthermore, another need was the development of a middleware, in order to support developers design and implement persuasive mobile apps. Indeed, using a persuasive middleware, developers could concentrate themselves design a mobile application from a high level, instead of getting into details about theories or models. In this way, they can save time, but at the same time they are sure to basing their applications above a solid theory.

A notable persuasive framework is Cicero, based on the ABM and developed by Professor Helal with the help of Antonello D’Aloia from the University of Bologna. I will explore further the middleware in Chapter 3, and as the main purpose of this thesis I will set up some improvements to it (Chapters 3 and 4).

Chapter 2 - Action-based Behavior Model and Cicero Middleware

As my first effort at the *Mobile and Pervasive Computing Lab* at the University of Florida I studied the ABM and Cicero, focusing on how improve the latter. In this Chapter, I will analyze both in the state in which I found them. Whereas, in Chapters 3 and 4 I will propose improves and modification to the structure of Cicero.

2.1. Action-based Behavior Model

Professor Helal and Doctor Lee proposed [4] [5] a model that is based on the collective knowledge gained by studying social and behavioral science theories. Specifically, the ABM was proposed as a persuasion template that computer scientists can understand and utilize.

Furthermore, as the name suggests, the ABM is based on sequence of actions that I'm going to describe. Firstly, the user's awareness about their conditions is increased by **informing** them about their current health status. This should increase their motivation by giving them the reasons for change. Next, **goals are set**. In this stage, several types of goal setting strategies are utilized: self-setting, assigned, participatory, as well as guided and group setting. This process allows the users to understand the details of the goals and the benefits of achieving them. Then, **users are educated** about how to achieve those goals. The next stage in the model is **reminding** the users to **act** toward the goals. Even if the users are highly motivated and are capable to act and achieve the goals, they may simply forget. There are two manifestations of the reminding process. The first manifestation reminds the users to get started acting towards the goal. The second reminder strategy is deployed gradually, informing users of their progress towards their goals. The final step is **rewarding** based on the achievement progress. Rewarding can be intrinsic (e.g., praise), extrinsic (e.g., gifts, credit, gift cards) or virtual reward (e.g. virtual credit as well practiced in game).

In Figure 3 each step is represented as a rectangle. The model partitions the system into a cyber system and a set of user actions. The cyber component is further divided into cyber sensing and cyber influence, to sense and influence exactly the set of actions prescribed by our model.

In the ABM, cyber sensors learn about the user's actions or inactions, and in general senses any relevant vital and status information. It also learns once initially about the user's profile and preferences.

Cyber influence, on the other hand, is where technological channels are used to deliver and affect controlled persuasion. Actions in the model consist of human actions (solid rectangles) and cyber actions (dotted rectangles).

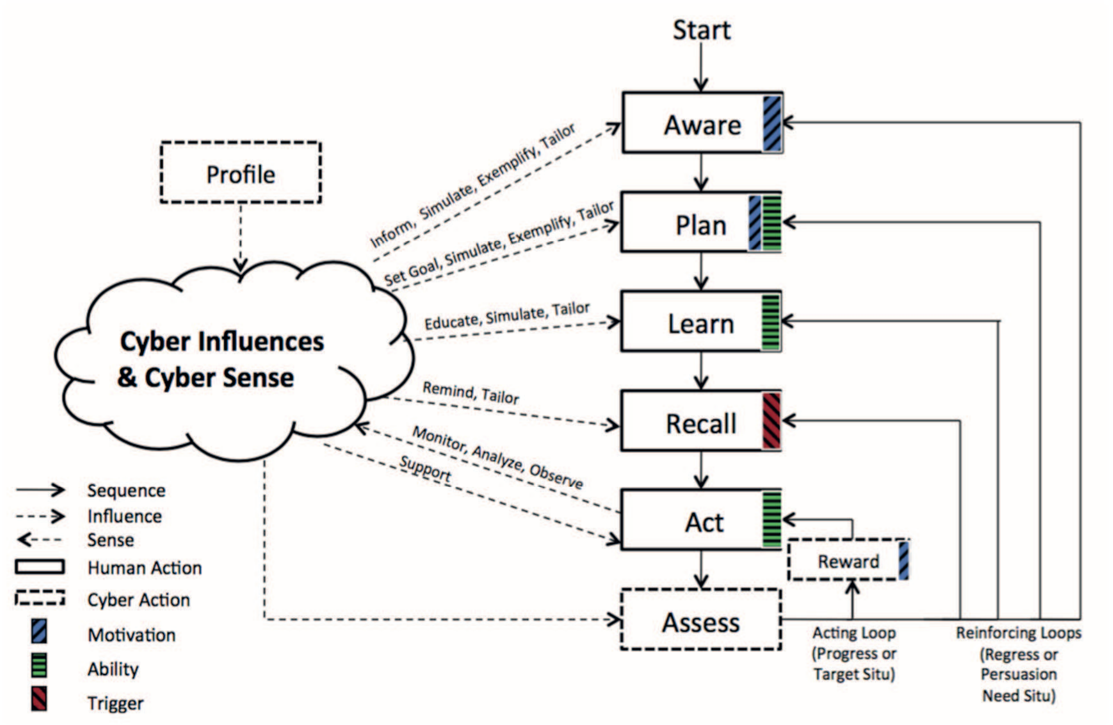

Start
Aware
Plan
Learn
Recall
Act
Reward
Assess
Profile
Cyber Influences & Cyber Sense
Inform, Simulate, Exemplify, Tailor
Set Goal, Simulate, Exemplify, Tailor
Educate, Simulate, Tailor
Remind, Tailor
Monitor, Analyze, Observe
Support
Acting Loop (Progress or Target Situ)
Reinforcing Loops (Regress or Persuasion Need Situ)
Sequence
Influence
Sense
Human Action
Cyber Action
Motivation
Ability
Trigger

Figure 3: The Action-Based Behavior Model

Moreover, the ABM relies on situation in order to detect persuasion needs and keep track of changing needs as well as changes in how a user responds to persuasive influence, especially in the Assess cyber action. Indeed, after the initial rounds of acting, the Assess evaluates the achievements of the goals and either rewards or rolls the user back to the appropriate action based on the achievement and deficit of each action. In order to do that, the Assess need a mechanism to detect changes in both persuasion needs and the user's responses to persuasive influences. Professor Helal and Doctor Lee decided to use situations for the latter purpose [6].

2.1.1. Situation

They define situation informally as a triplet of a user's activities, a set of device actions, and contexts over a period of time. Figure 4 shows two examples of situations. The first demonstrates the situation for "check glucose level successfully," which consists of a triplet of a device's action (glucose meter senses glucose level), a user's activity (measuring glucose level), and a context (numeric value of glucose level). The second example represents the situation for "fail to check glucose level," defined by a device's action (glucose meter generates an error code), a user's activity (measuring glucose level), and a context (the error code).

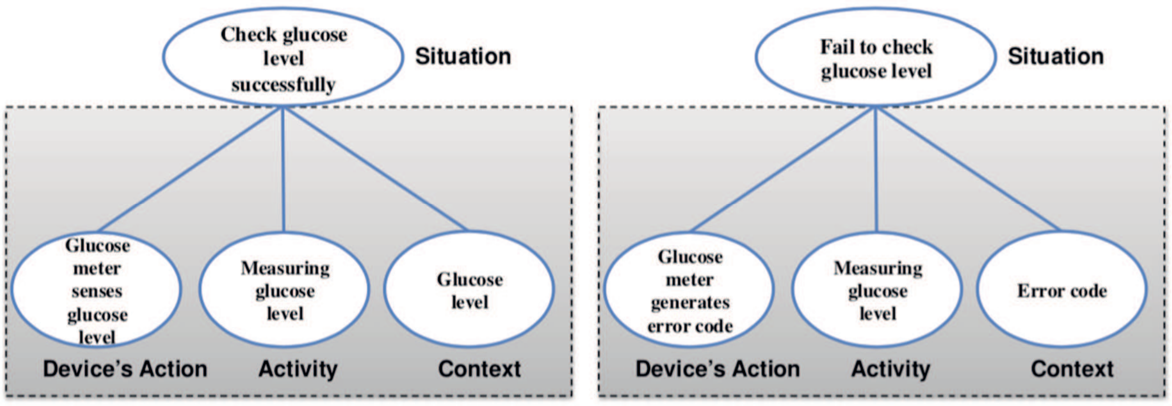


Figure 4: Two examples of situations

It is clear that while context is essential and important, it is not sufficient to characterize different user behavior responses in a persuasive system. Also, even though activity provides direct information of the user behavior response, it is not adequate alone and cannot be relied upon exclusively due to inherent inaccuracy of the activity recognition algorithms. In most existing definitions of situations, indirect sensing through sentience abstractions (e.g., activities, context, phenomena, and events) is used to infer user situations, but direct cybernetics such as device interactions are not used to measure or recognize situations.

Alongside this informal definition, Professor Helal and Doctor Lee propose a formal description of situation and its components.

2.1.2. *User Activity: formal definition*

User Activity is defined as the activity set related to a goal A = {A_1, A_2, … A_n}. The user activity further divides into active activity and passive activity. Active activity (A_a) is defined as the activity caused by the user himself and this could cause devices' passive actions when the active activity is related to the devices. Passive activity (A_p) is defined as the activity that is not caused by the user but other influences including devices' active actions. This can be denoted by

$$\begin{cases} A_a \;\; if \;\; A_k \rightarrow D_i^j \\ A_p \;\; if \;\; D_i^j \rightarrow A_k \end{cases}$$

A_a: $active\ activity$

A_p: $passive\ activity$

A_k: $certain\ activity$

D_i^j: $devices'actions$

2.1.3. *Devices and Devices' Actions: formal definition*

Device are presented by a device set D = {D_1, D_2,…,D_n}. The actions of a device are donated by D^j_i={ D^j_1, D^j_2,…, D^j_n}, where *j* represents device index and *i* action index. Those device's actions are categorized into active actions that affect a user's activity and passive actions that are affected by a user's activity. This concept is defined as follows:

$$D_i^j = \begin{cases} D_i^a \;\; if \;\; D_i^j \rightarrow A_k \\ D_i^p \;\; if \;\; A_k \rightarrow D_i^j \end{cases}$$

D_i^a: $active\ action$

D_i^p: $passive\ action$

2.1.4. *Context: formal definition*

Context is defined as the set of contexts related to a goal, where C={C_1, C_2,…,C_n}. Devices' actions could produce some information, which could be context. Thus, the device and context have a producer-consumer relationship.

2.1.5. *Situation: formal definition*

Situation is defined as a triplet of a user's activity, devices' actions and contexts, all of which are related to a specific goal and it is denoted by $\mathrm{S} = (\mathrm{D}, \mathrm{A}, \mathrm{C})$. In some cases, it is not possible to define one or two elements among S, thus a situation is allowed to be defined by one or more elements of a user's activity. A situation can be classified as an active situation or a passive situation. An active situation affects a user to change behavior and requires devices' active actions. An active situation is denoted by $\mathrm{AS} = (\mathrm{D}^{\mathrm{a}}, \mathrm{A}, \mathrm{C})$. A passive situation is used to express persuasion need situation, and evaluate a user's reaction from the active situation. Passive action is defined as $\mathrm{PS} = (\mathrm{D}^{\mathrm{p}}, \mathrm{A}, \mathrm{C})$.

2.2. *Situation-based Assess Tree*

Relying on situations just described, in [6] Professor Helal and Doctor Lee proposed a new structure called Situation-based Assess Tree (SAT), and used as an assessment instrument to implement the Assess step in the ABM. As shown in Figure 5, SAT is a five-level tree in which the root represents the Assess cyber action.

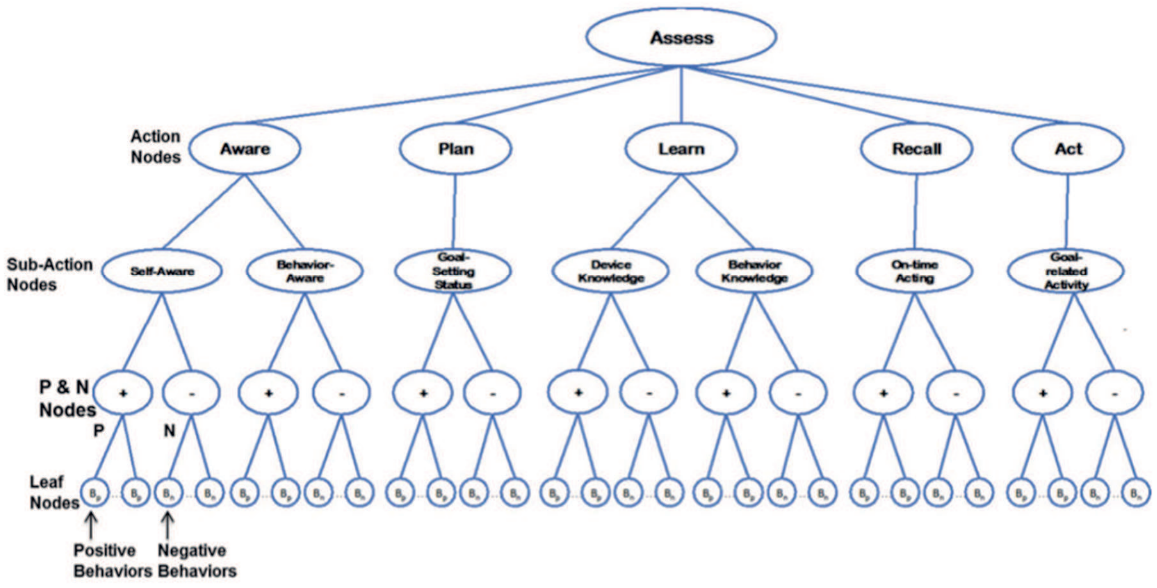


Figure 5: Situation-based Assess Tree

In order to better understand the Cicero Middleware explained in Section 2.3, I'm going to dig into detail of the SAT's composition and execution, because the latters are fundamental in the middleware's architecture and implementation. Under the root node, nodes at the second level are action nodes designed to match ABM's human actions. The third level refers to subaction nodes which further delineate specific elements of each ABM action in the second level. For instance, the Learn action has two subactions: one related to learning about what constitutes a good or bad behavior, and another that pertains to learning about relevant devices. It should be noted that while subactions are not part of ABM, they are designed to implement the model, and there could be more subactions possible for each of the ABM human actions. Next, each subaction node has one P-Node as a left child (denoted +) and one N-Node as a right child (denoted –), which, respectively, represent positive and negative behavior evaluator of that subaction node. By using positive and negative behaviors, subaction nodes are able to measure the net behavior. Collectively, P- and N-Nodes make up the fourth level of SAT. Finally, the fifth level consists of leaf nodes which are positive or negative behavior definitions. Leaf nodes must be properly attached to the intended P- or N-Nodes of subactions. For instance, a negative behavior leaf node "*Fail to Check Glucose Level*" (referring Figure 4) must be attached to an N-Node and not to a P-Node. Further, it must be attached specifically to the N-Node of the "*Device Knowledge*" subaction.

Furthermore, the power of this structure to implement the Assess step is the simplicity and intuitiveness with which it is possible evaluate this step. Indeed, simply by configuring and repeatedly evaluating SAT, the Assess cyber action in ABM (Figure 3) is evaluated. The SAT root value is the outcome of the Assess action which measures the achievements of the goals and either rewards the user or provides reinforcement back to the appropriate action. Rewards and reinforcements provide the necessary persuasion and support for the user to accomplish the set goals. In addition to utilizing the SAT root value for reinforcement, the user is informed and is made aware of her status in terms of every model action.

Concerning the SAT evaluation, it is worth mention that SAT is executed using SAT operators. Professor Helal and Doctor Lee defined several operators with several purposes in mind. Some act as comparators to capture the net effect of positive and negative behaviors. Some act as accumulators of various behaviors (e.g., in action and subaction nodes). Others act as diagnostic and analytic tools of various behaviors and assessment result. Overall, operators act as information fusion networks in which influential situations are identified and propagated over the tree to generate an assessment.

Moreover, SAT operators are divided into four different types:

- assessing operators, used to assess a user behavior;
- diagnostic operators, that can be applied to diagnose which behavior contributes to assessment value of the specific node;
- propagating operators, that can be exploited to control and limit transferring data to the upper nodes;
- analytic operators, that can be utilized to check the adequacy of registered positive and negative behaviors.

Beside this classification, each level of nodes has its own set of operators. In the Table 3, I list all operators proposed in [6], grouping them by the applicable node, and providing a brief description of each one of them.

Table 3: SAT Operators

Type	Operator	Description	Applicable Node
Assessing	$g(p)$	Calculate positive behavior value	Leaf Node
	$g(n)$	Calculate negative behavior value	
Propagating	F_A	Forward the behavior ID and value of leaf node all the way to the action node	
	F_P	Forward the behavior ID and $g(p)$ to the P-Node	
	F_N	Forward the behavior ID and $g(n)$ to the N-Node	
Analytic	R_P	Calculate correlation coefficient between $g(p)$ and Act value	
	R_N	Calculate correlation coefficient between $g(n)$ and Act value	
Assessing	$f(p)$	Calculate P-Node value	P-Node
Propagating	F_S	Forward the behavior IDs and $f(p)$ to the Subaction Node	
Assessing	$f(n)$	Calculate N-Node value	N-Node
Propagating	F_S	Forward the behavior IDs and $f(n)$ to the Subaction Node	
Assessing	I	Integrate one node value with the other node value	(Sub)Action Node
Propagating	C_S	Forward node value if it is smaller than threshold	
	C_B	Forward node value if it is bigger than threshold	
	C_R	Forward node value if it is bigger than first threshold and is smaller than second threshold.	
	C_U	Forward bigger value among node values	
	C_L	Forward smaller value among node values	
	F_A	Forward the behavior IDs and values to the action node	Subaction
	F_R	Forward the behavior IDs and values to the root node	Action Node
Diagnostic	U	Union behavior IDs and values into behavior set	All Nodes

2.3. Cicero Middleware

After studying and understanding the ABM and the SAT, my work at the Persuasive Lab at the University of Florida consisted in analyzing the current implementation of the middleware named Cicero, and figuring out improvement and optimization to it. In this subsection, I'm going to describe Cicero, paying particular attention on its architecture and implementation.

Cicero was initially developed [7] at the Persuasive Lab at the University of Florida, under the supervision of Professor Helal. On first analysis, Cicero is a middleware solution to support developers design and implement persuasive mobile apps. Based on the ABM previously described, Cicero provides developers with powerful class libraries and collaboration methodology to streamline the development of mobile persuasive apps without requiring a steep knowledge of behavior science theory or venturing into domain-specific knowledge and artifacts. Cicero guides the developers in following the ABM steps, provides APIs for cyber sense and cyber influence, and embodies the necessary model computations including measuring end-user compliance and response to influence and persuasion. Cicero also facilitates the engagement of domain experts in a clearly defined collaborative role.

2.3.1. Cicero Model

To design a mobile technology translation of the ABM, its cyber sense, cyber influence, and its Assess Tree algorithm, the cyberspace and its devices was limited to Android smartphones, tablets, and Android Wear devices. This is a feasible assumption, because Cicero aims to reach Android developer. Therefore, the ABM was reduced and shrunk to this context. Moreover, the set of available sensors on these devices are identified and explicitly made accessible to our middleware. Similarly, following a pyramid design methodology as shown in Figure 6, a rich set of potential activities and contexts are defined and explicitly supported in the middleware. Together, they constitute the second layer of the pyramid. It is important to note that information about activities could be used to recognize applicable contexts (hence the larger representation of Activity compared to Context in this layer). Finally, in the top of the pyramid, there is a programmable situations structure that may utilize any of the predefined contexts, activities or device interactions that are supported. Situations are key to implementing the Assess step in the ABM. It is also worth noting that device interaction is co-located with the situation layer because interactions with end users (e.g., through screen views, flash LED, vibration, or integrated speakers) are often coordinated based on materialized situations.

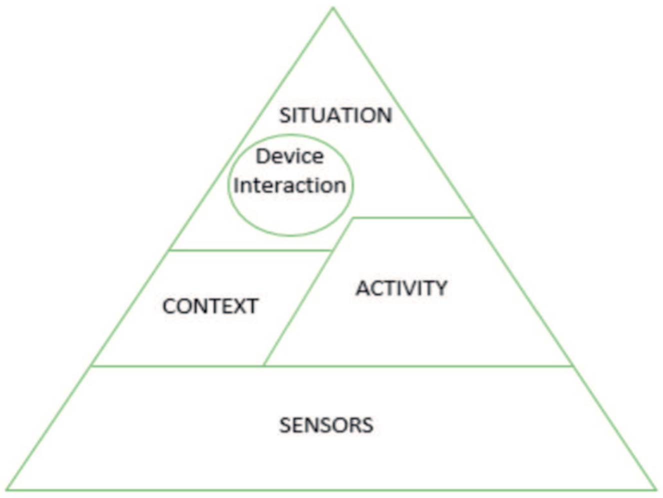


Figure 6: Cicero Design Pyramid

Now, I am going to analyze further each component of the pyramid, in order to have a complete overview of Cicero, fundamental before going into criticality.

Speaking about sensors, most Android smartphones and smartwatches have built-in sensors capable of providing high precision and accuracy raw data to measure motion, orientation, low-level activities, and various environmental conditions. In order to map such sensors into Cicero model, they are divided into three broad categories, following the Android subdivision [8]. These categories are the following: motion sensors (used to measure acceleration and rotational forces along three axes, such as accelerometers, gyroscopes, and vector sensors), environmental sensors (used to measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity), and position sensors (used to measure the physical position of a device, such as GPS or Wi-Fi).

Considering activities, Cicero maximizes the leverage of existing Android features especially Google Play Services, by offering rich APIs such as ActivityRecognitionAPI [9] and the FusedLocationProviderAPI [10] for managing motion and position sensors. Combined, they help recognize several user activities including still, tilting (the device angle relative to gravity change significantly), on foot, walking, running, on bicycle, in vehicle, and unknown. In order to support a broader set of activities, Cicero uses additional libraries and APIs offered by the Android platform including SensorManager [11], which is used to manage environment sensors, TelephonyManager [12], which is used to access to information about the telephony services on the device, and MediaRecorder [13] used to detect sounds around the user. Relying on just described components and on its API, Cicero is able to detect natively the following activities: *"walking", "running", "still", "calling", "going to <Location>", "being at <Location>", and "near <Location>"*. Furthermore, developers are provided with a very simple interface to register and detect occurrences of activities. Moreover, advanced developers are able to implement the detection of new activity, extending the middleware.

Concerning context, the latter is determined by combining data that derive from different sensors. I denote gathered sensor data with the symbol $SE_k(x)$ which represents the sensed data by the kth sensor. Using AND, OR, and NOT operators, it is possible define the following three context models [7]:

1. Locational Model, which is based on the locations in which the device is located. Hence, using this model a context is a combining of sensed data in different locations. This model would show how the changing of context in the space, in particular the progress that it has depending on different locations;

2. Snapshot Model, in which context combines sensor data collected in different points in time;
3. Value Model, that is similar to the first two models, but defined over ranges of values rather than specific single sensor values. This model is more practical to use in real-world applications. Context under Value Model can be expressed as follows: $C = SE_1(v_a, v_b)\ op\ SE_2(v_a, v_b)\ op \dots op\ SE_k(v_a, v_b)$, where v_a and v_b represent the two extremities of the range of a given sensor SE_i. As implementation choice, Cicero uses this model, because it is more complete, it presents continue values, and it allows the use of different kinds of filter.

Finally considering situations, by recognizing different activities, contexts, and device actions, it is possible to recognize the occurrence of predefined situations that will enable the SAT assessment algorithm and the ABM Access step. Cicero utilizes a simple quorum-based decision method to recognize situations: if two or more elements of a situation are true (out of three, i.e., device action, context, and activity), the situation is considered to have occurred [14].

2.3.2. Cicero architecture and implementation

After the analysis of the model that reduces the ABM to something implementable in Android devices, I am going to introduce the main characteristics of Cicero architecture and its components. This analysis that I made in my first month at the University of Florida was crucial in the second part of my job. In fact, after I thoroughly explored the current implementation at the time of my arrival, I found some criticality that I will present in the Section 3.1, while in this subsection I am going to introduce an overview of the Cicero's architecture and implementation.

Cicero works on mobile devices in a layer underlying persuasive apps. For the developer, Cicero is a middleware and a tool that simplifies the development process of persuasive applications. For the end-user, Cicero is a run-time under which Cicero-based apps execute. Developers can create persuasive applications by setting cyber sense and cyber influence and, in general, by setting and programming each step in ABM. This includes setting goals and contents relative to the Aware and *Learn* steps, setting suggestions and reminders relative to the *Recall* step, and setting *Rewards* based on available incentives. End users are transparently guided by ABM and its steps: they can use the app by setting goals, checking their own progress, interacting with reminders, and obtaining rewards.

Following the latter guidelines, an architecture, shown in Figure 7, was created, and it consists of two main managers: sentience manager at the bottom and cyber manager, the rectangle at the top of the figure. The former is concerned with sensing in general and allows the developer to select contexts and activities to monitor. The cyber manager works to facilitate the developer's work to set and program the various steps in ABM. In addition, it is concerned with recognizing situations and monitoring the progression of the various steps. Periodically, the cyber manager would evaluate how the user conforms to the settings of the model, and progresses towards goals through the SAT algorithm. SAT evaluations are done and all others processing are done locally on the hosting mobile device. This was a critical point, further analyzed in Section 3.1.

Concerning the **Sentience Manager**, the latter accesses and gathers sensor data using listeners and filters data in order to evaluate the achievement of goals. Sentience Manager implements the *Act* step of the ABM, thus it is initialized and processed by the cyber manager. In order to manage the different sensors, the Sentience Manager is further divided into several components:

- LocationScan, which is used to manage position sensors. In particular, this component is based on FusedLocationProviderAPI [10], an efficient module integrated in Google Play Services.
- MotionScan, formerly ActivityScan, which is related to the motion sensors. I renamed this component in order to avoid confusion with activities and situations. This module

can detect in background the user’s activities using ActivityRecognitionAPI [9] offered by Google Play.

- EnvironmentScan, which is used to manage environmental sensors, such as amplitude or lightness.
- SocialScan, which is related to social activities such as making or receiving phone calls, and the frequency of such activities.
- TimeService, that is an Android Service used to manage and filter sensed data about time context, such duration, or frequency.
- SensorService, that is another Android Service used to collect gathered data from sensors.

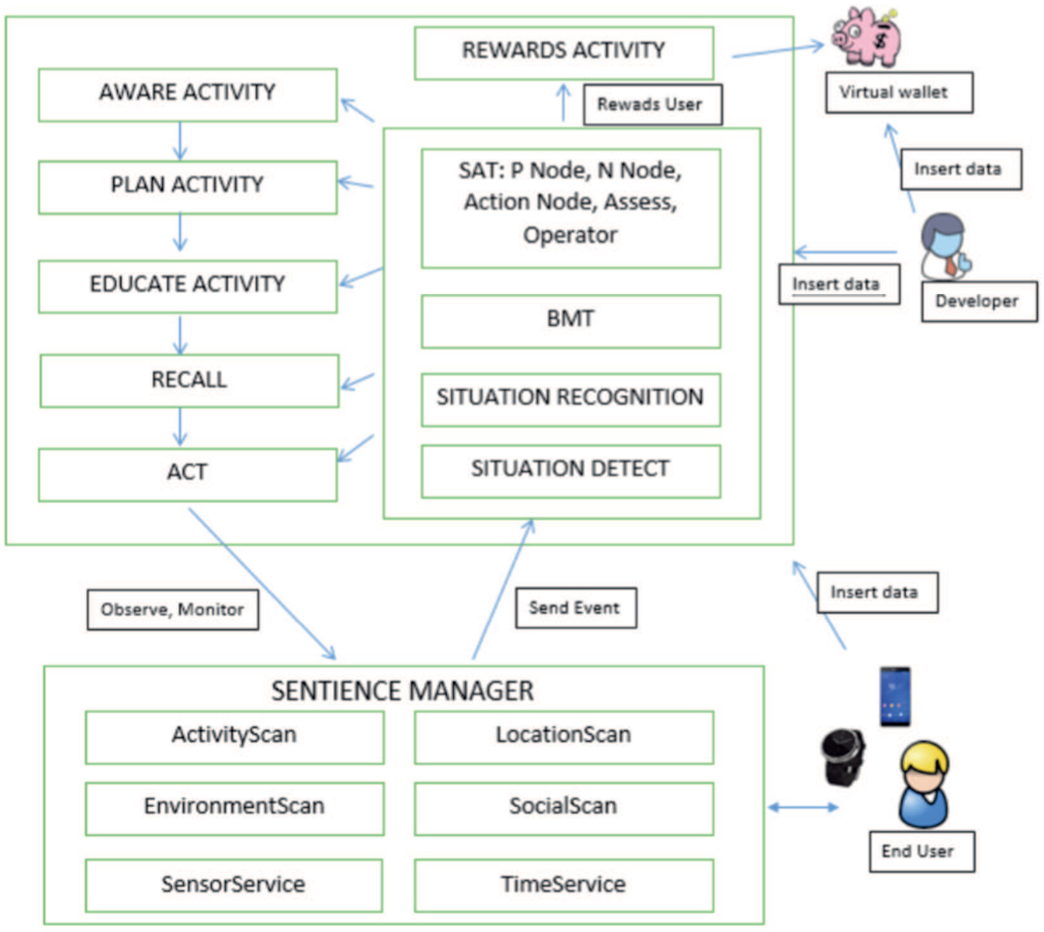


Figure 7: Cicero Architecture

These last two Android Services run in the background, and they allow filtering data and the assessment of the goals achievement. Unfortunately, at the current situation it is possible to sense only one context for each Android Service. It is clear that this is a very big limitation, and I will go into details in Section 3.1. Moreover, another important responsibility of the Sentience Manager is sending events to the cyber manager in order to detect the different situations that drive the SAT algorithm evaluation.

Regarding the **Cyber Manager**, it implements all the ABM steps except for the *Act* (which is managed by the Sentience Manager). For *Aware*, *Plan*, *Learn*, and *Recall* steps the developer has to create own activities, but it is hard to insert them into Cicero architecture. That made Cicero not very modular and rigid to developers' changes and customization. Clearly, also this point will be detailed in the Section 3.1. Moreover, *Aware* and *Learn* steps should inform and educate the end user, thus they have to show influential contents. Cicero facilitates for such content to be placed by a domain expert into a shared Google Drive folder accessible to the user through the app. This enables the collaboration and participation of the domain experts, for example physical therapists or doctors, who know best what content are appropriate, and who are capable of using something as simple as Google Drive (but not any more complex than that). Unfortunately, this simplicity is also a disadvantage, because with Google Drive the domain experts are able only to share something with an end user, but the former are not able to monitoring the patience through Cicero. Therefore, the collaboration is limited, and it is one way. The *Plan* step, depending on the goal-setting theory, should guide the user during the configuration of variables that characterize the goals. After the Plan step is set, Cyber Manager initializes the Sentience Manager using the scanning classes and Sentience Manager's own initialization methods. Moreover, reminding the user is a very important aspect of persuasive apps so the user can behave and converge towards the set goals. Thus, *Recall* step is realized by Cicero using Android Notification triggered according to set schedules at configuration time, and by SAT through reinforcement loops. Cicero sends notifications only when the device is active to ensure user's attention is secured. When the device is locked the notification is queued and sent as soon as the user unlocks the device. The last module of the Cyber Manager is the *Assess* step that implements the SAT algorithm to evaluate the changes in user's behavior, compliance, and response to the persuasive influences of the various steps. Cicero contains Java classes that implement the SAT shown in Figure 5. In order to create and populate this tree, domain experts describe the behaviors, while the developers use Cicero to implement behavior nodes into situations using context, activity and device interactions.

These are the main characteristics of Cicero middleware's architecture, and some implementation details. In the next section, I am going to describe related work about persuasive middleware, and differences compared to Cicero.

2.4. Related Work

Despite of the increasing number of applications with persuasion features, only few middleware and frameworks are written to support and streamline their development. Moreover, some middleware is specific for mobile crowd sensing lacking underlying persuasion theory. In this section, I analyze related work on persuasion middleware, paying specific attention to their cyber sense and cyber influence capabilities, and to their differences and similarities with Cicero middleware.

Code In The Air (CITA) [15] [16] is a system developed in the Networks and Mobile Systems group at MIT that simplifies the development of complex tasking applications using a web interface. CITA enables non-expert end users to express easily simple tasks on their smartphones, and more sophisticated developers to code complex tasks by writing purely server-side scripts in JavaScript. It is worth mentioning that CITA is able to sense and recognize a number of low-level activities such as *isWalking*, *isDriving*, *enterPlace*, and *leavePlace*. In addition, CITA allows developers and users to compose lower-level activities using logical predicates to create high-level activities. Regarding the ability to persuade the user, CITA provides limited support in this regard in the form of triggers and reminders. It uses an asynchronous message delivery service, which is used in order to trigger the user to fulfill some tasks.

Funf [17] initially developed at the MIT Media Lab is another extensible sensing and data processing framework for Android mobile devices. It aims to help developers create applications that need mobile sensing easily without having to access low-level APIs. While CITA and Funf support cyber sense, they both lack cyber influence and persuasion features. There are also some commercial applications similar to CITA and Funf, most notably Tasker [18], which allows users to perform tasks based on contexts such as time, date, location, event, input gestures, or using information provided by other applications. Thus, while CITA, Funf and Tasker all provide cyber sense capabilities, Tasker provides richer sentience as it is capable of providing sensor values in addition to contexts which are richer, modular, and extensible by developers.

The Pervasive Middleware for Activity Recognition (PEMAR) framework [19] developed at the University of Missouri aims to increase the level of physical activity by creating a middleware for active games on mobile devices. PEMAR is able to recognize human motions from the perspectives of human activities and their contexts, but it also lacks an underlying theory of persuasion, though it uses games as a method of persuasion. Moreover, PEMAR is implemented as an applicable middleware for understanding human motion and mapping it to other forms of gaming applications. To ensure wide applicability, authors provide the real-time analytics solution for activity recognition with ensuring continuous learning and availability of activity models with new datasets. They also include modeling of contexts in contrast to traditional approaches that deal mainly with the raw data of motions (such as video data or motion data). In addition, authors construct a shareable repository for activity recognition by providing online learning features that stream new data. This ensures enhancement of the accuracy of recognition as well as reduces the burden of gesture training by providing an activity library.

The Framework for Intelligent Healthcare Self-Management [20] developed at the National University of Sciences & Technology of Islamabad combines ubiquitous and social computing as persuasion media. The authors developed an application and a social web site for diabetes self-management using different persuasive strategies, like rewards, effort messages and achievements. In addition, the app promotes social interactions with friends and familiars, through which the patient may be persuaded. The framework is not however a middleware aimed for use by other developers, and it is not based on ABM, but on a different persuasive model called *Monitoring-Assessment Model* (MAM). MAM consists of eight stages that are listed below.

- *Profiling*, a stage required to get user information for creating user profile.
- *Target Behavior*: the system needs to identify a target behavior for change which can be inferred by the user profile or recommended by the experts.
- *Monitoring*: the user's entire activities are observed for gathering of data related to physical status (vital signs, glucose level, etc.), activities (exercise, food consumption, social interaction), and behavior (mood, mental status and feelings).
- *Analysis:* the stage of analysis is the core of MAM. In this stage, the gathered data from monitoring stage is analyzed and form the basis for the user's *health profile*.
- *Intervention:* the health profile allows a doctor, nurse, or a care taker to keep track of patient's health. It also allows doctors to be able to recommend them in case of emerging problems.
- *Persuasion:* if the analysis shows that the user is lacking in some aspect (e.g., exercise, diet, or medication), then he or she is motivated to improve on the lacking behavior. The user can be either persuaded by the system, by the expert or by the social context.
- *Action:* a persuasion strategy is meant to induce some behavior change in the user. If it is successful, the user will take some measures or corrective actions.
- *Assessment:* In this last phase, an assessment of the user's activities, behavior, and physical status is carried out. A comparison is made against whether the analysis and intervention have been successfully applied through some persuasion strategies to materialize any behavior change in the user and whether any action was performed as a

result. Based on the outcome of the assessment, either a new target behavior is identified for the user or the previously targeted behavior is strengthened by applying additional persuasive strategies.

The Patient-Clinician-Designer (PCD) framework [21] developed by Marcu, Bardram, and Gabrielli provides guidelines for overcoming the challenges of designing applications for mental illness through sensitivity to the needs of these patients, and the equal involvement of both patients and clinicians in the design process. In particular, the framework applies a user-centered design process that is especially sensitive to the complexity of the mental illness, the difficulty of treatment, its stigma, and the goals of patients and clinicians. Therefore, PCD framework is targeted both to end users and application designer, but it is very limited in scope to monitoring systems specialized for mental illnesses.

Google recently released Google Fit [22] - an open platform that allows users to control their fitness data, and supports developers build health and fitness apps. It also targets manufacturers and aims to influence their future device designs to take advantage of Google Fit. With this platform and with Google Play services location APIs [23], developers should be able to easily create health and wellness applications. Nevertheless, without proper support for persuasion, developers have to create a lot of code from scratch, which is time consuming and involves a steep learning curve.

MoST [24] is a smartphone sensing library developed at the University of Bologna. It is very similar to Google Play services location APIs, but it is open source, lightweight, modular, and efficient especially regarding battery use. In particular, MoST offers an activity detection pipeline that provides an efficient and flexible component to collect high-level inference about the real world condition of users. Indeed, MoST provides a uniform abstraction layer to access smartphone hardware and logical sensors (e.g., accelerometer, gyroscope, GPS, app networking statistics, battery level, etc.) and eases the burden on application developers by taking into account concurrency issues due to access to shared resources, thus making sensing un-intrusive and without negative impact on user experience. All collected raw data can be directly accessed by client applications, but MoST also implements signal processing and machine learning technique to processes raw data into high-level inferences about user physical activity and audio context (i.e., detecting human voice, noise, and silence).

Finally, DailyAlert [25] is a generic mobile persuasion toolkit for smartphones, which generalizes and automates mobile persuasion as a service for mobile applications. It follows the thin-client design to reduce power consumption on the users' devices and to simplify integration with mobile applications. Although DailyAlert is a complete persuasive toolkit, similar to our proposed ABM middleware, it is not based on an explicit theoretical foundation or models.

Chapter 3 - Introducing a new cloud-based architecture

At the time of my arrival at the Persuasive Laboratory of the University of Florida, Cicero was in an early state of progression. After I studied the ABM theory, and after I analyzed its implementation, I figured out some improvements in order to refine some criticalities that I found. In this Chapter, I am going to describe these criticalities, and introduce new models and a brand new cloud-based architecture that aims to solve some issue of the previous version. Basing on this new architecture, I developed a new version of Cicero, for sake of clarity called Cloud-Based Cicero (CBC). Then, in the Chapter 4 I will go into details of implementation facts of this new architecture.

3.1. Critical points of Cicero

First of all, an important issue with the previous version was that Cicero was able to sense only a behavior at time. That was very limiting because it made Cicero **mono sensing**. In other words, Cicero was not able to support every persuasive application that requires more than one behavior: as a matter of fact, every real application was not supported. In order to solve this problem, and in order to make possible the multi sensing, I modified the Cicero's architecture. Indeed, I created a new structure that supports the sensing of more behaviors concurrently, relying on a single Android Service that owns a poll of object, each one has the task to sense a single behavior. That new architecture is further analyzed in the Section 3.2.

Secondly, another issue found in Cicero was that it was designed only for Android smartphones, or tablets. Since my arrival at the Persuasive Lab, that was viewed as a possible improvement. In fact, as first thought Cicero was a middleware able to sense from multiple devices at the same time, for example colleting location from a smartphone, and at the same time recognizing the activity from a smartwatch. Nevertheless, for sake of simplicity, and because of the very early state of the implementation, the middleware was able to act only a **mono-device sensing**. Then, I decided to implement the sensing on Android smartwatches, leaving the door open to easily integrate the sensing on further devices. This improvement created a new question: how can a middleware that maintains the SAT locally be split up into more devices, each one sensing separately, but contributing to the same SAT? In order to answer this question, I created a new cloud-based architecture, in which the SAT is maintained remotely, but every device that would like to join the sensing can sense a behavior locally. This new architecture will be explained into details in the Section 3.3. Moreover, in the Section 3.4 I will analyze a new model and a communication protocol, both specific for Android smartwatches.

Thirdly, the participation of the **domain expert** was not as strong as expected. Indeed, as mentioned in Section 2.3, in Cicero the domain expert was involved in two ways: helping the developers building the SAT, and sharing useful files with the patient via Google Drive. While the first kind of involvement is fundamental and needed, the second one is not very expressive, and primarily it is not biunique. In fact, end users are able to access and consult documents loaded by domain experts, but latters cannot watch over improvements of the formers. This could be an important feature for a persuasive middleware, helping the domain expert to be more engaged, and in a better way. Unfortunately, adding this new feature in Cicero was very difficult and time-consuming, but with the new cloud-based architecture this improvement is easier. Indeed, we can imagine a domain expert as a new special node, which cannot sense, but is able to look at the remote SAT in order to see improvements of the patients. Furthermore, in this way the domain expert is also able to modify patients' SAT, even create them for the first time. That allows domain experts to accomplish their first involvement in Cicero in a simpler way, because they are guided remotely by a GUI with this specific task. In the Section 3.3 I will also analyze the involvement of the domain expert in this new cloud-based architecture, while in Section 4.1.3 I will show the application that I build in order to let domain experts interact with SAT.

Finally, Cicero needed a general **refactoring**, concerning both the way it is integrated and used by developers, and how it is structured internally. In fact, for sake of simplicity, Cicero was built up along with a case of study, but the latters are strongly coupled. Notably, the development of the Android Activities needed from the ABM (e.g. the Activity that inform the user about the importance of a healthy behavior – *Aware* step) is responsibility of the persuasive developer. Nevertheless, Cicero included each of these Activities. It is clear that this approach, useful for the first implementation, could not be exploited in a final version made available for other developers. Furthermore, also the way to share Cicero was under review. In fact, we needed a self-contained middleware, which could be shared with other developers. The latters should be responsible only to create application-specific Activities, and to inform Cicero which one should be used for each ABM step. This new structure will be further investigated in Sections 4.1.5 and 4.1.6, after analyzing implementation choices that are fundamental to understand the new organization.

Regarding the internal structure, in Cicero there are no defined packages, or a specific separation between groups of Classes. Thus, a general restyling was necessary in order to make easier to understand, modify, and extend the code. This change was mainly a refactoring of the existing code, moving Java classes into a packages' hierarchy that reflects the ABM structure. Indeed, I divided the code into two main packages, one for the cyber sense, and one for the cyber influence. However, this new structure was further changed in order to implement the new architecture explained in Section 3.3, but the main structure remains the same, although in a distributed context.

3.2. Sensing Architecture

In CBC, I introduced a new architecture that allows sensing concurrently several behaviors. The main components of this architecture are summarized in the Figure 8. Going into details, in the cyber sensing, the main entity is the *Sentience Manager* (SM). This manager relays on an Android Service running in background, named *Sentience Manager Service* (SMS). SMS is the only Service running, and it has to manage the multi sensing. In order to do that, I introduce a new concept: the *Sentience Object* (SO). SO is an object that maps a particular sensing (e.g. there is a SO dedicated to sensing if the user is running in a duration context), and it has to manage and store the state of the sensing. Indeed, SO gathers information from sensors, obtained using the proper Google Play Service (e.g. ActivityRecognitionAPI), or low level methods. In order to support multi sensing, SMS owns a pool of SO, each one is actually gathering data about a different behavior. This pool is named *Sentience Pool* (SP).

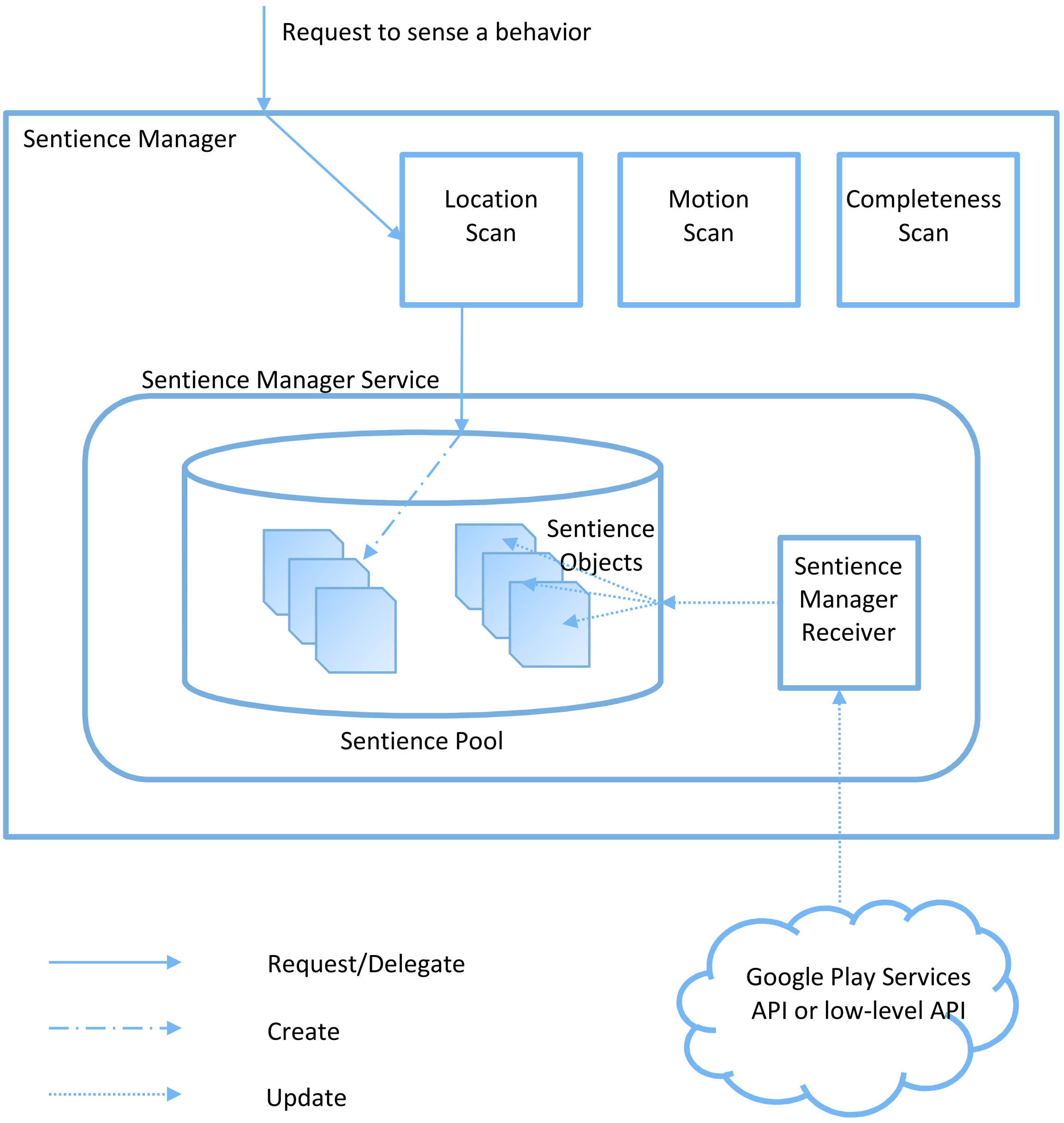

Request to sense a behavior
Sentience Manager
Location Scan
Motion Scan
Completeness Scan
Sentience Manager Service
Sentience Objects
Sentience Manager Receiver
Sentience Pool
Request/Delegate
Create
Update
Google Play Services API or low-level API

Figure 8: Sentience Manager architecture

It is also worth noting that there is one other group of Classes inside the SM: the *Scan*s. These objects are useful delegates, employed to SM in order to add a SO inside the pool. Indeed, as shown in the Figure 8, when a new request to start sensing a behavior reaches SM, the latter delegates the task to the right *Scan* object. For example, SM uses *Location Scan* when a behavior based on a location arrived (e.g. "be at the gym"). Then, the creation of a new SO (and the insertion into SP) is up to the *Scan*.

Furthermore, SM owns also a broadcast receiver that waits update from the Google Play Services API, or from low-level APIs. This component is called *Sentience Manager Receiver* (SMR). After receiving an update, SMR informs the pool, and then all SO associated with this update (e.g. all the SO waiting for an update about a location are informed when a new location is detected by the FusedLocationProviderAPI). That behavior is also shown in the Figure 8.

3.3. Situation-based Assess as a Service

Another significant enhancement to Cicero is the new cloud-based architecture, which results in the CBC. This new architecture, shown in Figure 9, splits the main cyber influence component away from devices, bringing the former to the cloud. Going into detail, the main component of the cyber influence is the *Cyber Manager* (CM). Moreover, the CM is deployed into a cloud platform, and it contains the SAT symbolizing user behavioral response. In this way, SAT is unique for every user, even if they have multiple devices. However, the cyber sensing remains locally on each node in which the end user is running CBC.

Splitting the two main components of Cicero (CM and SM) is needed to decouple sensing to assessing, but it creates new challenges regarding the communication between them. In fact, in Cicero these two components were both local, and they could communicate with simple method calls. Now, in the new CBC architecture CM is on a cloud platform, while SM is not necessarily unique, but it is certainly on devices. It is clear that this decoupling in the space makes the communication much harder.

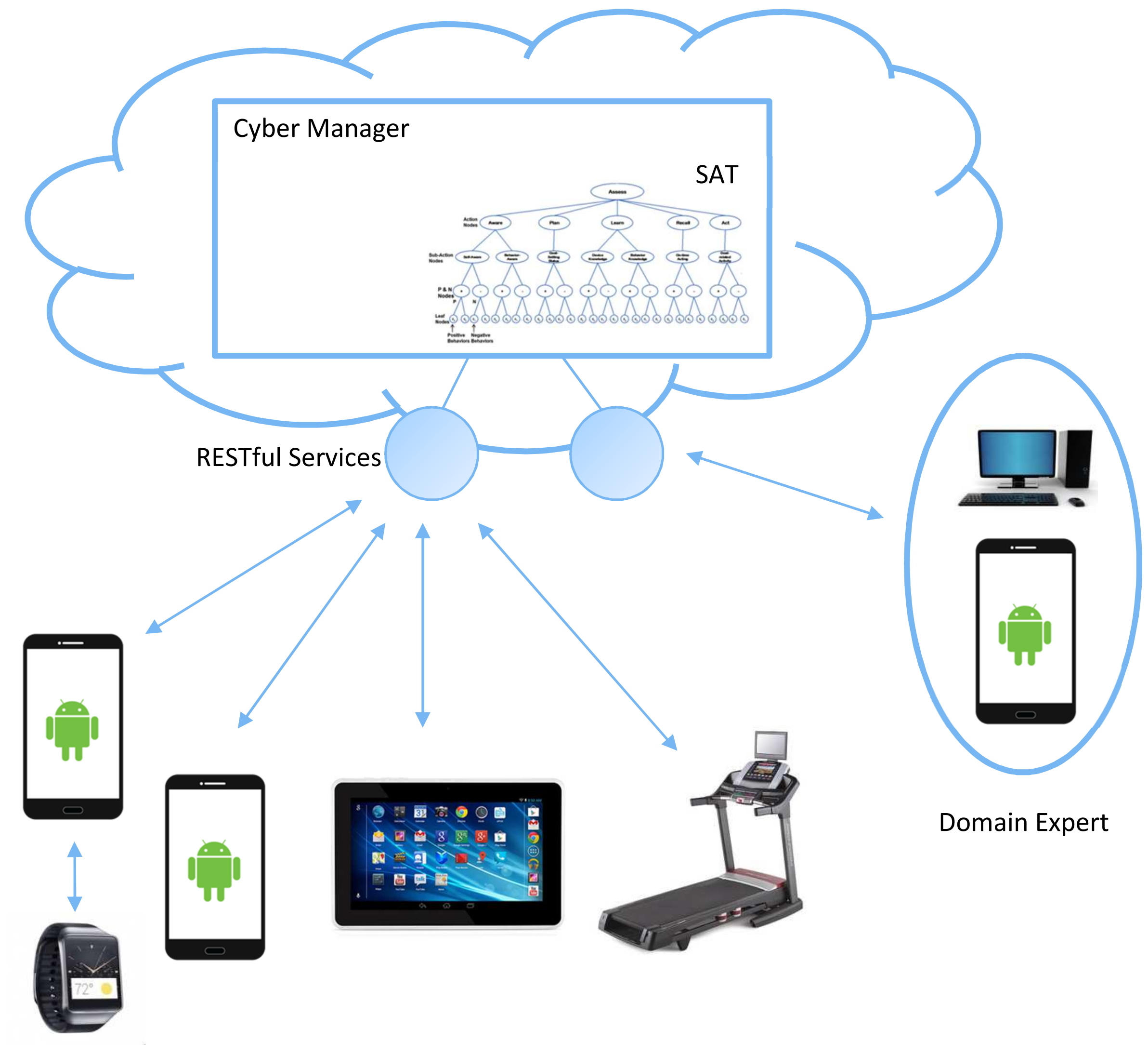


Figure 9: Cloud-based architecture with multiple devices

Nevertheless, analyzing the Cicero implementation, I found that only few communications are needed between CM and SM. Indeed, we can divide these interactions into three main categories:

- Events produced by the occurrence of a particular situation sensed on devices: these events are generated into devices, but it is up to the SAT to manage them;
- Events and results generated by the SAT after the execution of the assess: now the SAT is on the cloud, then we need a way to inform the local part of CBC on devices about results of assesses;

- Requests of start sensing a particular behavior: on the previous version, when the SAT was created, CM informed SM in order to let the latter starting sensing the desired behavior.

In order to enable these kinds of communications, I adapted the new architecture in the following way. First of all, I designed some RESTful services that allow remote devices to communicate with the CM on cloud. Particularly, when an event occurs on a sensing node (e.g. on a smartphone), the local part of Cicero calls a RESTful service via HTTP request, or using an out-of-box client library, in order to inform CM of that event. In this way, I made an issue a new feature. In fact, with this simple assumption, I opened Cicero to sense potentially on every device able to make HTTP requests.

Furthermore, in order to enable the second kind of communication, I designed a message system similar to the common push notifications model. In fact, after an assess CM sends a message to each SM in sensing devices previously registered. In this way, SMs are updates but they are decoupled from the unique CM and SAT.

Finally, I resolve the third kind of communication with a new model of registration. Indeed, in the new architecture the sensing nodes have to notify the cloud-based CM of the willing of start sensing. After that, the CM informs the new sensing device about which behaviors he needs to sense. With new simple protocol, I remove the need of a communication from the CM to the SM, because now the initiative is up to the SM. Moreover, we can deal with the eventuality in which the SAT is not already created by the domain expert. In that case, the local SM asks for behaviors to sense, and CM simply responds that there is nothing to sense.

Furthermore, a new need of communication arose. In fact, we need a set of services designed for domain experts. Indeed, latters need to interact with the remote CM from their smartphone, or PC, and they need a mechanism to inspect, and modify the remote SAT. It is clear that these needs require a communication between domain experts' devices and the cloud-based CM. I resolved this problem creating separate RESTful services (specific to the domain experts) with which they can creates nodes to add to the SAT, or simply observe the progress of the patient.

All these types of communications are shown in the Figure 9 as simple or dashed arrows. I did not write about implementation choices, but I only described models and architectures. I will analyze into detail the implementation in Chapter 4, explaining the technologies chosen for implementing CBC architecture.

Finally, it is worth to note that, even if CBC middleware is running in every node, we have two diversifications. Indeed, we have a cloud-side component, deployed on a cloud-based platform, offering assess and domain-expert support as services. Besides, there is another component embedded in every application relying on CBC. The latter part has the task of sensing, and communicating whit the former some important events that happen locally. In other words, CBC project is divided into two further projects: one project for the cloud-based part (which services are deployed, and world-wide accessible), and another project that will be distributed to developers (that will add it as dependence to the persuasive application's project). I will further investigate projects' structure, and implementation choices in Section 4.1.

3.4. Smartwatches integration

In the Figure 9, a sample Android smartwatch is shown linked to another smartphone, and not directly to the cloud-based CM. This inhomogeneity is caused by the different nature of a smartwatch compared to a smartphone. In fact, sadly an Android smartwatch is not able to execute an HTTP request to internet, expect for the case in which it has a Wi-Fi connection, and it is not connected via Bluetooth to the linked handheld device. Since this is a remote eventuality, in most of cases the smartwatch cannot communicate directly to the remote CM.

Considering this, we cannot deal smartwatches like other nodes. In fact, they can sense, but not communicate directly with the remote CM. Therefore, I created a new communication protocol between the smartwatch, the handheld, and CM. I called this protocol Smartwatch-To-Cloud Communication Protocol (SCCP). With SCCP, I enabled communication between the smartwatch and CM, letting the smartwatch communicate with the CM through the connected handheld that behaves like a bridge, resending messages from the watch towards CM, and vice versa.

Chapter 4 - Implementation Details and Experimental Results

In Chapter 3 I analyzed the new cloud-based architecture, focusing on new models, and how I improved Cicero. Besides, in the Section 4.1 I'm going to detail implementation facts of CBC, reasoning about implementation choices. Then, in Section 4.2 I will show some experimental results about CBC.

4.1. Cloud-Based Cicero Implementation

As introduced in Chapter 3.3, CBC runs on every node that joined sensing. Nevertheless, there are two main diversifications of it. Indeed, CBC is composed by a cloud-based part of the middleware, which is accessible from every device through exposed APIs, and by a client-side part residing on each node.

In order to implement this structure, I created three different Android Studio projects. As detailed in Section 4.1.1, the first one is a Google App Engine Project, and it is named App Engine Project (AEP). AEP contains the implementation of the cloud part of CBC, and it will be deployed on the Google cloud platform.

The second project is an Android library, and it represents the client-side part of CBC. This project, named Android Library Project (ALP), is included as a dependency by every persuasive Android application's project that developers create. I'm going to describe ALP in Section 4.1.2.

The last one, named Cicero Commons Project (CCP), is a regular Java library, and it contains the minimum amount of common Classes that should be accessible from each one of two previously-mentioned projects. Indeed, CCP contains the definition of Situation and Behavior, their events, and classes useful for the creation of users' profile (explained in Section 4.1.3). These Classes are needed both by AEP and ALP, thus both of latters include CCP as dependency. It is worth to note that CCP is a regular Java library. In this way, it can be included as dependency both by Android and non-Android (e.g. AEP) projects.

Moreover, in Section 4.1.3 I will present the Android application developed for domain experts, and in Section 4.1.4 I will detail the protocol that enable the communication between smartwatches and CM on the cloud.

Finally, in Section 4.1.5 I am going to show how CBC is developed on the cloud platform, and how it is shared to developers. Furthermore, in Section 4.1.6 I will present a use case, going into detail on how developers can exploit CBC in order to develop persuasive applications.

4.1.1. Server-side Implementation

In order to implement the server-side part of CBC, the first decision I made was the choice of the most suitable cloud platform. I considered that CBC needs two main kinds of communication between cloud platform and devices, as described in Section 3.3. In particular, CBC needs a way with which devices can inform CM about new events, and a tool that enable cloud-to-devices communication (like push notification) in order to inform users' applications about assess' results.

In order to fulfill these needs, I decided to use **Google App Engine** (GAE) [26] as cloud platform. GAE is a cloud-based platform (*Platform as a Service*) for building scalable web applications and mobile backends. Among other features, GAE offers two useful tools that CBC can exploit: **Google Cloud Endpoints** (GCE) [27], and **Google Cloud Messaging** (GCM) [28].

Going into details, GCE consists of tools, libraries and capabilities that allow generating APIs and client libraries from an App Engine application, to simplify client access to data from other applications. Endpoints makes easier to create a web backend for web clients and mobile clients such as Android or Apple's iOS. Moreover, as shown in Figure 10, the API backend is an App Engine app that performs business logic and other functions for Android and iOS clients, as well as JavaScript web clients. The functionality of the backend is made available to clients through

Endpoints, which exposes an API that clients can call. Regarding CBC, using GCE lets our model to be open to future extensions (e.g. sensing on non-Android devices), uncoupling backend implementation to devices' ones. In fact, in principle GCE is accessible from every device is able to perform an HTTP request, though it is commonly used in conjunction with client-side libraries developed for Android, iOS, or JavaScript clients.

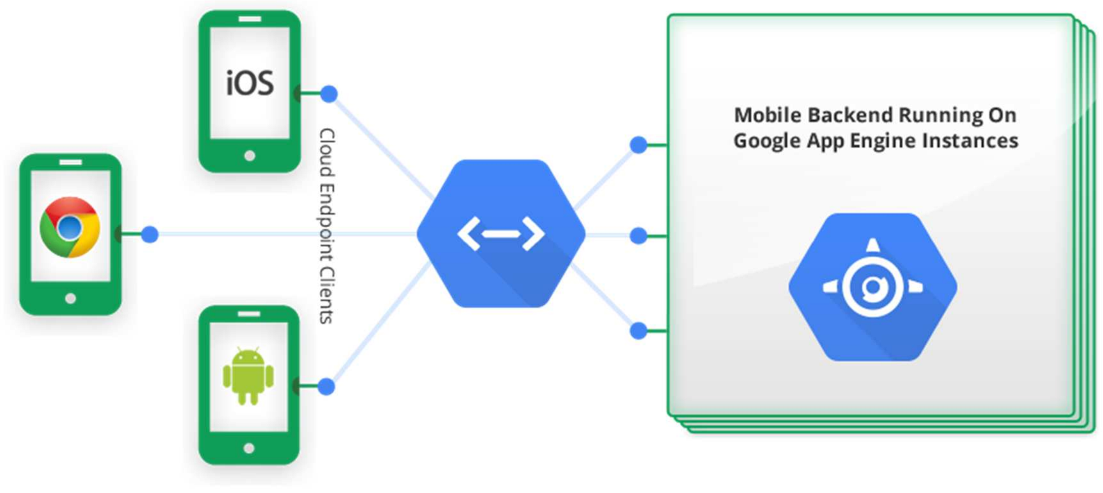


Figure 10: GCE APIs architecture

Besides, GCM is a mobile service developed by Google that enables developers to send notification data or information from GAE instances to applications that target the Google Android Operating System, as well as applications or extensions developed for the Google Chrome internet browser. Moreover, GCM has the ability to send push notifications, through a feature named Downstream Messaging (DM), with which GCM provides a reliable and battery-efficient connection between server and devices. Thus, CBC can rely on DM in order to send a message to devices after completing an assessment, specifying the result, and actions needed.

Moreover, GAE also provides a service (named App Engine Cron Service), that allows to configure regularly scheduled tasks that operate at defined times or regular intervals. This service could be critical in CBC, because it needs to perform an assessment of SAT with a fixed rate (i.e.

every `ASSESS_WINDOW` milliseconds). Conversely, Cicero exploits local timer, but in GAE environment it is not possible to use them. Moreover, Cron Service requires only few details in order to be implemented. Indeed, it needs a descriptor file (`cron.xml`), and a Servlet (in CBC, `AssessServlet`) at which an HTTP request will be send every time the Cron is executed. In particular, in the `doGet` method of this Servlet, CBC assesses SAT, and it sends back to sensing devices a message containing the result of the assess, if necessary.

In light of these tools and GAE's completeness, I decided to use it as cloud-based platform in which developing server-side part of CBC. Then, I created a new App Engine Project (AEP), and I moved CM into this project. In this way, CM is running in GAE backend instance. In next two subsections, I am going to details implementation's details about Endpoints that I created (Section 4.1.1.1), and about the new GCM system (Section 4.1.1.2).

Finally, in Section 4.1.1.3 I am going to introduce improvements made to SAT in order to make it more usable, and more consistent with ABM.

4.1.1.1. Google Cloud Endpoints implemented in Cloud-based Cicero

In order to communicate with CM, I created two main GCE, one for normal sensing, and one that enables communication between CM and domain experts. The former, named `CiceroEndpoint` (CE), exposes some useful public methods, then automatically converted into remote APIs by Android Studio and GAE tools, along with an Android client library that help accessing these APIs from an Android device. In particular, CE has methods that help remote sensing devices interfacing with CM. One of this methods is named `sendSituationEvent`, and it lets sensing devices propagate a Situation Event occurred locally towards CM residing into GAE backend. Indeed, in Cicero Situation Events generated by sensing a particular behavior are propagated locally through listeners (i.e. Observer pattern), and thanks to `SituationEventManager` (SEM), until they reach interested behavior nodes. Nevertheless, in CBC that is not possible, because nodes (and SAT) are stored remotely into the CM, but events are initially generated on devices. In order to solve this problem, I created a new Class in AEP named `RemoteSituationEventManager` (RSEM). Besides, I modified the behavior of SEM (residing in ALP): when it handles a Situation Event, instead of propagate the event locally, it makes a remote call to the `sendSituationEvent` API, using the auto-generated Android libraries, and passing the event as argument. Then, CE informs RSEM which notify every behaviors node previously registered. With this new protocol, I created a sort of bridge between local Situation events propagation, and the remote one, in which the bridge between SEM and RSEM is a remote call to CE. At this point, `sendSituationEvent` method has the only task to inform RSEM of the received Situation Event (in Code 1). It is also worth to note that the marshalling and the unmarshalling is up to the auto-generated libraries, and RSEM receives a `SituationEvent` object.

```java
@ApiMethod(name = "sendSituationEvent")
public void sendSituationEvent(SituationEvent situationEvent) {
        CyberManager.getInstance()
        .getRemoteSituationEventManager()
        .fireSituationEvent(
                situationEvent.getSituation(),
                situationEvent.getDate(),
                situationEvent.getValue()
        );
}
```

Code 1: `sendSituationEvent` method

Others notable methods of CE are `idsToSense`, with which a sensing device is able to retrieve a list of IDs that it has to sense, and `behaviorToSense`, that returns the specified behavior passing its ID. In fact, as mentioned in Section 3.3, a sensing mobile node needs these methods in order to start sensing. Indeed, the mobile device requests the list of behaviors' IDs it needs to sense, then obtains the Behavior instance copy useful in order to understand how sense it (e.g. a location required sensing device's location, and not user motion, nor light sensor's data), and finally it can start sensing.

The second notable GCE implemented in CBC is needed to communicate with domain experts' application that I am going to detail in Section 4.1.3. Indeed, this Endpoint, named `DomainExpertEndpoint` (DEE), is reserved for domain experts' interactions, in order to create separation with CE, mainly for security reasons. In fact, DEE exposes all APIs remotely accessible by domain experts' applications, in order to inspect and modify SAT, and containing sensitive information. Going into details, there is a method (then mapped as API) called `profile`. When remotely invoked, this method serializes the current user profile, and sends it back to domain expert's application which can use and analyze it after deserialization. Moreover, using the API named `tree`, it is possible to retrieve the current situation of the SAT. Finally, DEE exposes other methods useful to add behaviors to SAT that help domain experts tailoring SAT to users' changes.

Finally, AEP contains an Endpoint named `RegistrationEndpoint`. It is used to register a device in the GCM system, and it will be explained further in Section 4.1.1.2.

4.1.1.2. Google Cloud Messaging implementation

In order to enable a push notification system in CBC, I implemented GCM. In this subsection, I'm going to detail which Classes were added in order to let client-side part of CBC receive downstream messages.

First of all, the mobile sensing device needs to register with GCM. In particular, according to GCM specifications, client applications must register with GCM in order to verify that they can send and receive messages. In this process, the client obtains a unique registration ID (a String) and passes it to the GAE server, which stores the ID. The registration ID exchanged in this process is the same client app instance identifier that the GAE server uses to send messages back to the particular client.

Going into details, to register with GCM:

1. The client application obtains a registration token through GCM Service [29], in particular using `register` method of `GoogleCloudMessaging` Class. Moreover, this method returns the registration ID uniquely identifying the client.
2. The client application passes the registration token to the app server, remotely calling the API `register` (coded in the `registerDevice` method) of `RegistrationEndpoint`, and passing the registration ID just obtained.
3. The application server saves the registration ID. Storing this ID lets CBC to send a message to every device sensing that registered itself.

Most of the effort in this handshake protocol is borne by the CBC's client-side part, and it will be explained into details in Section 4.1.2.2. Regarding the AEP, the GCM implementation concerns the `registerDevice` method of `RegistrationEndpoint` which code is shown in Code 2. It is worth noting that the static method *`ofy`* return an `ObjectifyService` Class, that is part of the Objectify library [30]. This library offers a simple convenient interface to the Google App Engine datastore. Using Objectify developers can simply save or load an entity, only calling a method. In this case, `registerDevice` method saves a `RegistrationRecord` instance containing the registration ID received as parameter. Furthermore, Objectify is also used in AEP in order to persist SAT and the users' profile. In fact, because of its implementation, GAE backend creates an instance of CM in the cloud platform, but it deallocates it after a while if no other requests to GCE, web pages, nor Servlet have come. Then, in order to prevent loss of information about SAT or users' profile, CBC stores them in the GAE datastore, exploiting Objectify to easily save, load, and delete entities. In order to persist instances of `AssesTree` and `Profile` Classes, I created `AssesTreeDTO` and `ProfileDTO`, two Classes that own a serialized instance of `AssesTree` or `Profile`. Moreover, they are annotated with Objectify's `Entity` annotation. In this way, it is possible to persist them in GAE datastore, and, when a new instance of GAE backend is created, retrieve the related SAT and users' profile, preventing loss of information.

```java
/**
 * Register a device to the backend
 *
 * @param regId The Google Cloud Messaging registration Id to add
 */
@ApiMethod(name = "register")
public void registerDevice(@Named("regId") String regId) {
    if (findRecord(regId) != null) {
        log.info("Device " + regId + " already registered");
        return;
    }
    RegistrationRecord record = new RegistrationRecord();
    record.setRegId(regId);
    ofy().save().entity(record).now();
}
```

Code 2: `registerDevice` method of `RegistrationEndpoint`

Returning to GCM implementation, after a sensing device is registered, CM can send it downstream messages. In particular, after an assessment of SAT, a new message is created, and it is sent back to all registered devices. Moreover, we add a parameter to the message that specify which Activity the device needs to show. This parameter is obtained inspecting the values of SAT's nodes. In particular, if an Action node's value is lower than a fixed threshold, then the corresponding Activity needs to be shown to the user. For example, if a user lacks of awareness about his-or-her condition, it is likely that the value of the Aware node in the SAT is lower than 0.5, and then CBC needs to show to the user on his-or-her Android smartphone an Activity showing some links to articles, or videos that make him or her more aware. In this example, if the Aware node's value is lower than 0.5, then the GCM message will contain a parameter with name "`c_afterAssess`", and value "`aware`". When the message will reach sensing devices, the corresponding Android Activity will be shown. I am going into details about how the client-side part of CBC do that in the Section 4.1.2.2.

```
public static void sendAfterAssessMessage() {
        Message msg =
                new Message.Builder().addData(
                        "c_afterAssess",
                        getAfterAssessString()
                ).build();
        Sender sender = new Sender(API_KEY);
        List<RegistrationRecord> records =
                ofy().load().type(RegistrationRecord.class).list();
        for (RegistrationRecord record : records) {
                Result result=sender.send(msg, record.getRegId(), 5);
        }
}

public static String getAfterAssessString() {
    if
(CyberManager.getInstance().getAssessTree().getNodeFromPath("assess.aware
").getValue()<0.5) {
        return  "aware";
    } else if
(CyberManager.getInstance().getAssessTree().getNodeFromPath("assess.plan"
).getValue()<1) {
        return  "plan";
    } else if
(CyberManager.getInstance().getAssessTree().getNodeFromPath("assess.learn
").getValue()<0.5) {
        return  "learn";
    } else if
(CyberManager.getInstance().getAssessTree().getNodeFromPath("assess.recal
l").getValue()<0.5) {
        return  "recall";
    } else {
        return  "none";
    }
}
```

Code 3: Send a GCM message back to sensing devices

4.1.1.3. SAT improvements

Besides implementations of GCE and GCM, AEP differs from Cicero's CM module also for same improvements made to SAT structure, and to behavior.

First of all, I need a way to identify each leaf node (behavior) in SAT, in order to avoid confusion for sensing devices that request behaviors to sense, and that need to retrieve them. For that reason, I created a path-like IDs system to identify every node (not only leaves) inside SAT, in which every node is identified by its name in ABM, or by a unique number (behaviors). For example, the behavior leaf node denoted by ID 2 under the P Node of Self-Aware Sub-Action is identified by the path "`assess.aware.selfaware.pnode.2`". It is worth noting that the leaf node's ID is chosen as an increasing integer, unique among behaviors inside parent P-or-N Node. Then, in the previous example, there are at least two behaviors owned by Sel-Aware's P Node. Moreover, I created some methods useful to retrieve a node's instance from its path (e.g. `getNodeFromPath(String path)`). In addition to these methods, I also added a method to obtain ID from Node's instances. Indeed, I added a method named `getPath` inside base `Node` Class. This method returns the ID in a recursive way. In fact, it adds the current ID to the path of the parent obtained calling the same method over it.

Secondly, in Cicero SAT was not homogeneous. Indeed, the top part of the tree was formed by nodes, but leaves were behavior. This heterogeneity caused a lot of troubles dealing with SAT using recursion and inheritance, because some nodes had other nodes as child, but someone had behaviors. In order to do that, I created a new implementation for the SAT. Indeed, I added a new Class named `BehaviorNode`, and extending the base Class called `Node`. This kind of node represents leaves, and it is an adapter for a behavior. In fact, each instance of `BehaviorNode` owns a behavior, besides the current value inherited from the base Class. Using this new Class resolves also another problem of Cicero: previously it was needed using a map in order to track current value because Behavior Class does not have a field for the current value, but now in CBC `BehaviorNode` can deal with it easily. On the top of leaves node, there are P-or-N Node. In order to implement them, I created a new abstract class that extends `Node`, and it is named `PNBaseNode`. This Class contains useful methods in order to deal with `BehaviorNode` that are its child. For example, it contains a method that create and add a `BehaviorNode` directly from a `Behavior`'s instance. Moreover, I created two further Classes that extend `PNBaseNode`: `PNode`, and `NNode`. The latters are responsible to calculate the new value of a children node considering if the node represents positive (P) or negative (N) behaviors. Finally, all other nodes at the top of the tree are instances of `Node`, denoted by a value, a name (the last part of the path as previously described), a parent (that may be null if the node is the root or not inserted in the tree yet), and a list of child that are instances with Class `Node`, or its subclasses. In this way I created a more usable and accessible SAT.

The third enhancement to SAT is more linked with the new CBC model. In fact, it is not sufficient to completely describe a behavior indicate only the reference value. Indeed, the domain expert has to specify several information in addition to that value. For instance, if a domain expert wants to add a behavior that is "run 20 minutes every day", it is not sufficient indicate 20 minutes as reference value of a behavior of type duration, but it is also needed to specify 24 hours as a deadline for the duration. In fact, this additional information is useful to the context in order to know if the user has to run 20 minutes every day, every week, every month, etcetera. Moreover, it is possible that the behavior is incomplete without some information about the device action. For example, a domain expert can add a behavior like "be at gym twice a week", but he or she cannot specify the preferred gym for the user. In order to solve this problem, in CBC domain experts add a behavior that is incomplete, and then before start sensing users complete it adding information about device action. Then, the client part of CBC saves the new compete behavior in remote SAT, and starts sensing it. It is also worth noting that in the last example, in addition to the location information about context are needed: in particular, the deadline for the frequency similarly to what previously described (a week). Therefore, in order to solve these problems, I modified the `Behavior` Class adding two new fields: `contextAdditionalInfo`, and `detectAdditionalInfo`. Using them it is now possible to have a complete behavior, ready to be sensed, or stored.

4.1.2. Client-side Implementation

In addition to the server-side part, CBC is also formed by a client-side part that is developed inside ALP. Essentially, ALP is composed by the following three main components that I'm going to detail in next subsections: some Classes that extend Android `AsyncTask` (Section 4.1.2.1), Classes that allow implementing client part of GCM (Section 4.1.2.2), and Classes related to sensing (SM implementation detailed in Section 4.1.2.3).

4.1.2.1. `AsyncTask`*s Implementation*

ALP contains some Classes that extend Android `AsyncTask`. These Classes have the task to communicate with the Endpoints exposed by AEP. Since the communication is usually through the Web, it requires some milliseconds, or even seconds. In order to not block the user interface, CBC uses these Classes that execute the remote call in background. Depending on the particular `AsyncTask`, it may require to execute some actions in the GUI thread (e.g. close a loading pop-up, or launch an Activity). That can be done in the `onPostExecute` method that is executed in the main thread.

Classes that can be found in ALP are one for each API exposed by AEP, and they are implemented in a similar way. For instance, ALP contains `GcmRegistrationAsyncTask` that register a device to GCM system, `GetProfileAsyncTask` that returns the user's profile, and `GetTreeSituationAsyncTask` returning the current SAT situation. Moreover, the complete code of the latter `AsyncTask` can be found in Appendix A.1. `GetTreeSituationAsyncTask`.

4.1.2.2. Client part of GCM

As mentioned previously in Section 4.1.1.2, CBC uses GCM in order to send downstream messages. In addition to the server-side implementation of GCM, also ALP contains some Classes necessary to GCM. In particular, inside ALP there are some `AsyncTask` that have the task to register or unregister the sensing device in which ALP is running. Moreover, after the registration CBC needs to listen for future possible messages received from the remote CM. In order to do that, I implemented two Classes. The first one is `GcmBroadcastReceiver`. The latter is, as its name implies, a Broadcast Receiver, and it is registered inside the Android Manifest in order to handle messages sent from GCM to application. Inside the method `onReceive`, GcmBroadcastReceiver launch a second class that extends an IntentService, named GcmIntentService. This Class has the task to check if the received message contains an extra parameter named “c_afterAssess”, and if so launch the related Android Activity. A code snippet of the onHandleIntent method of GcmIntentService is shown in Code 4.

```
if (extras.containsKey("c_afterAssess")) {
    //assess occurred on remote: maybe we need to show an activity
    String c_afterAssess = extras.getString("c_afterAssess");
    if (c_afterAssess!=null && !c_afterAssess.equals("none") &&
Cicero.getInstance().isConnected() &&
!Cicero.getInstance().isShowingActivity()) {
        switch (c_afterAssess) {
            case "aware":
                Cicero.getInstance().launchAwareActivity(this);
                break;
            case "plan":
                Cicero.getInstance().launchPlanActivity(this);
                break;
            case "learn":
                Cicero.getInstance().launchLearnActivity(this);
                break;
            default:
                Toast.makeText(getApplicationContext(), "c_afterAssess
type not known", Toast.LENGTH_LONG).show();
                break;
        }
    }
}
```

Code 4: Snippet of onHandleIntent method of GcmIntentService

To sum up, after implementing these Classes, CM is now able to send downstream messages, letting sensing devices know which Activity they have to show after an assess of the SAT is completed.

4.1.2.3. Sentience Manager implementation

The last part of ALP is maybe the most important, but it is certainly fundamental for sensing. In fact, the package edu.ufl.cicero.sentience contains the implementation of the new CBC’s sensing architecture, already explained in Section 3.2.

Going into details, `SenitenceManager` Singleton Class (SM) is central in the implementation. In fact, SM acts as a façade to the sensing module. Indeed, all requests to start or stop sensing a specific behavior must pass through it. Moreover, SM owns four instances that are crucial. In fact, it owns an instance of `SentienceManagerService` (SMS) set after the latter is bounded, and one instance separately for `MotionScan` (MS), `LocationScan` (LS), `CompletenessScan` (CS) Classes. As mentioned in Section 3.2, these *Scans* are useful delegates that facilitate adding a new SO into SP. I'm going to detail these *Scans* later in this Section, after analyzing SP.

Regarding **SMS**, it has the task of receive updates from the Google Play Services APIs, or from low-level APIs, through a broadcast receiver called `SentienceManagerReceiver` (SMR). SMS also owns an instance of `SentiencePool` (SP). As I have already introduced in Section 3.2, SP realizes the contemporaneous sensing of more behaviors. In order to achieve that, SP owns three set of instances of three different types. The first one contains instances of `CompletenessSentenceObject` (CSO) Class. CSO, that implements `SentienceObject` (SO) interface, is a class that implements sensing a behavior that requires checking the completeness of it. In order to do that, CSO exposes a method, `notifyCompleteness(boolean complete)`, that launch a Situation Event when invoked. Moreover, CSO is the simplest SO, because it does not require a real sensing: when a user complete an action in the Android application (e.g. when user sets the location of her-or-his usual gym), all CSO inside SP's set are notified, and the one waiting for this completeness launches a Situation Event. The second set contains instances of Classes that implement the `MotionUpdateListener` (MUL) interface that extends SO. Conceptually, MUL objects are behaviors that are waiting for updates about the way user is moving (i.e. they are waiting an update from the Activity Recognition API). Similarly, the third set is formed by instances that implement `LocationUpdateListener` (LUL) interface that also extends SO. LUL objects represent SO that are waiting for updates about the location of the device (i.e. they are waiting updates from Fusion Location Provider API). Going into details, MUL and LUL are at the top of a hierarchy that allows SP to multiple sense behaviors. On the other side, at the bottom of the hierarchy there are twelve concrete Classes. Indeed, ALP contains one Class for each combination between contexts (actually six, three related to the clock subcategory – duration, frequency, exact time – and three about sensors – light, temperature, heart rate), and device actions that require sensing (actually two: location, and motion). Between interfaces and these twelve Classes, there are some abstract Classes used as support in the implementation (e.g. grouping common methods). Please refer to Appendix A.2. *SO Hierarchy* in order to see the complete hierarchy of SOs. Furthermore, exploiting this inheritance, SP can store in a single set all SOs that are waiting for particular updates about a context, without knowing something more about implementation, nor the way the update is dealt after SP notify it to the related SO. Returning to SP, when SMR receive an update, the latter notify the former calling one of methods shown in Code 5. It is worth to note that each SO is informed inside a different thread. Indeed, SP does not known how long is the elaboration of the update by every SO, so it is more prudent and efficient execute each elaboration in a separate thread than in a sequential way. Moreover, SP does not make any filtration, nor operation on the received update. Indeed, it is task of each SO deal with new data, also ignoring them if it not pertinent whit the related behavior.

Regarding ***Scans***, they are useful delegates that help SM to start sensing a behavior. Indeed, when SM needs to start sensing a new behavior, SM looks for which *Scan* is able to start sensing that particular behavior, invoking the static method named `canScan` along with the activity name of the behavior. When SM identifies which *Scan* between MS, LS, and CS is able to start sensing, the former delegates it to start sensing. Then, the delegated *Scan* bounds to SMS, and invokes its method `sense` along with the behavior instance. Using this instance, SMS is able to instantiate a new SO of the appropriated Class, and add it to the right Set inside SP.

```
public void receivedMotionUpdate(final String name) {
    for (final MotionUpdateListener listener: motionUpdateListeners) {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                listener.onMotionUpdate(name);
            }
        });
        t.run();
    }
}

public void receivedLocationUpdate(final String name, final Location
location) {
    for (final LocationUpdateListener listener: locationUpdateListeners){
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                listener.onLocationUpdate(name, location);
            }
        });
        t.run();
    }
}
```

Code 5: `SentiencePool`'s most important methods

4.1.3. Domain experts' Application

As previously introduced in Chapter 3.1, one critical point of Cicero is the unidirectional engagement of domain experts. In fact, the latters can share useful links or document with their patients, but they cannot have any feedback about improvements of the formers. Moreover, I have already detailed in the Chapter 3.3 how add this important feature inside the new CBC structure.

Entering into the implementation details, I created an Android application that may be bundled into CBC distribution as is, or with minimal changes in order to be tailored to particular domain's needs or experts' request. With this new application, named Domain Experts' Mobile Application (DEMA), the domain expert can now easily check assesses results in real time, modify the SAT according new needs (e.g. adding new behaviors with new the reference values), and track improvements throughout patient's history. And everything in an independent way, thus without informing the developers of the mobile application.

Moreover, DEMA is an Android application that presents three main functionalities: showing the current SAT situation, graphing the history for a particular level (for sake of simplicity, in Figure 11 is shown only the history of the root-level assess value, but the same graph could be made for any value in the Profile), and presenting an Activity with which the domain expert is able to add a new behavior (see Figure 12).

In order to let DEMA interact with CBC, the former uses the APIs exposed by DEE, previously detailed in Section 4.1.1.1. Furthermore, I added to the client-side part of CBC some Classes that extend Android `AsyncTask`. In Appendix A.1. `GetTreeSituationAsyncTask`, I show an example of these `AsyncTasks` (i.e. `GetTreeSituationAsyncTask`). These reusable Classes can be used in DEMA in order to communicate with the Endpoints. In this way, DEMA has only the task of present to the user the received data.

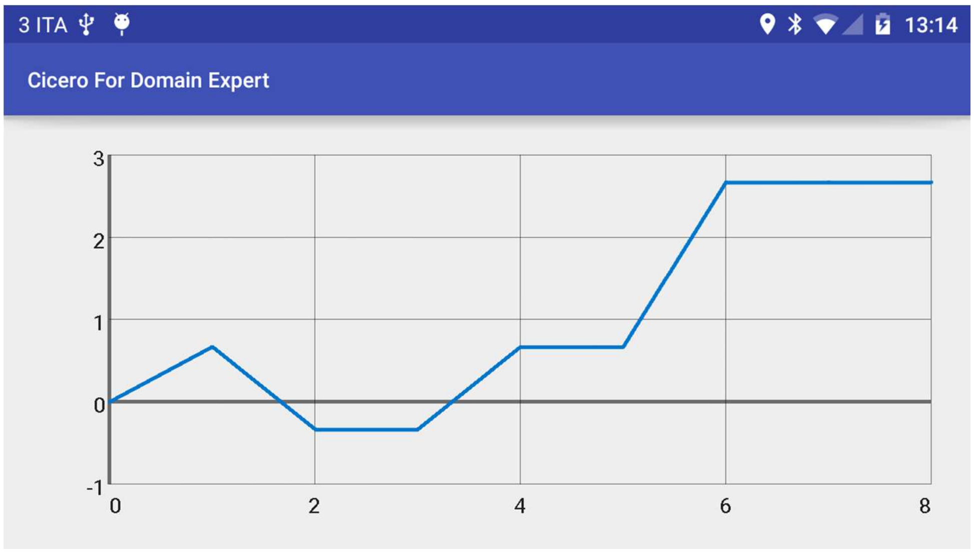


Figure 11: A graph with an example assess history shown by DEMA

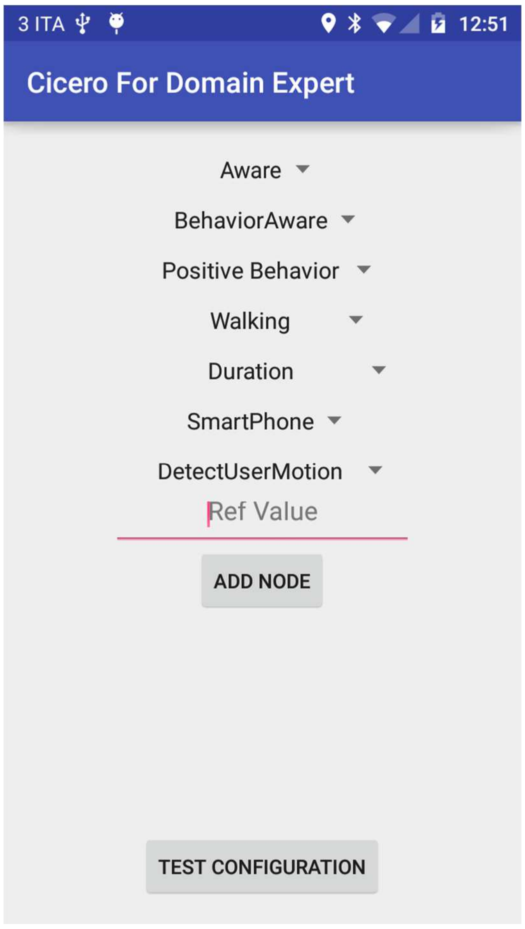


Figure 12: DEMA allows domain expert to add new behavior nodes in a very simple way

In order to make this possible, I improved the way Cicero profiles users. In fact, in Cicero profiling is not developed completely yet, but it is only outlined. Thus, I created a new Serializable Class called Profile owning a list of Assess Snapshots (AS) instances. Each AS represents the snapshot of the SAT made after an assessment. Using AS, Profile is able to maintain the full history of the SAT with a fixed frequency, that is the assess frequency (i.e. every `ASSESS_WINDOW` milliseconds). Moreover, every AS makes a complete and depth snapshot of the current situation, saving values for every SAT's level, and storing behaviors actually present. In this way, Profile owns the most complete history about patients, being able to use it in complex evaluations depending on the specific domain. Furthermore, I created a new singleton named Profile Manager (PM), residing on the cloud part of CBC, and accessible from Endpoints. PM owns and manages the Profile, taking care of its creation and maintenance, and offering a method called `profileCurrentSituation` useful in order to add a new AS.

4.1.4. Smartwatch-To-Cloud Communication Protocol

As introduced in Section 3.4, I create a new protocol (named SCCP) in order to let communicate smartwatches and CM that is running on GAE platform. In principle, every device with an Internet connection is able to communicate with CM through GCE and APIs exposed by AEP. Unfortunately, Android smartwatches are not able to execute HTTP requests in normal conditions. Indeed, an Android smartwatch is able to communicate directly to Internet only if it is not connected via Bluetooth with its handheld, but it is connected via Wi-Fi to an Access Point. Since Wi-Fi is not common among smartwatch devices on the market, this eventuality is not frequent, and in most of cases smartwatches are not able to execute an HTTP request.

In order to solve this problem, I created SCCP, a protocol that enable communication between smartwatches and CM through the handheld. Going into details, I take advantage that every sensing node base its application on top of ALP, and using SM as façade in order to start sensing a behavior. Indeed, when a persuasive application running on a handheld asks SM to start sensing, the latter requests a list of behaviors that it needs to sense to CM running on GAE platform. Besides, if there is a connected smartwatch, SM also requests a list of behaviors that smartwatch needs to sense (Step 1 in Figure 13 that shows SCCP). After obtained the list (Step 2), SM serializes it, and sends it to the connected smartwatch via MessageApi service [31] (Step 3). Using MessageApi, an Android device can send a message to a connected node (in this case a smartwatch) attaching an optional payload, and a path that uniquely identifies the message's action. In SCCP, SM prepares a message containing a serialized list of behaviors, and sends it to the connected smartwatch, using a path defined as a constant in ALP (`TO_WEAR_MESSAGE_PATH`).

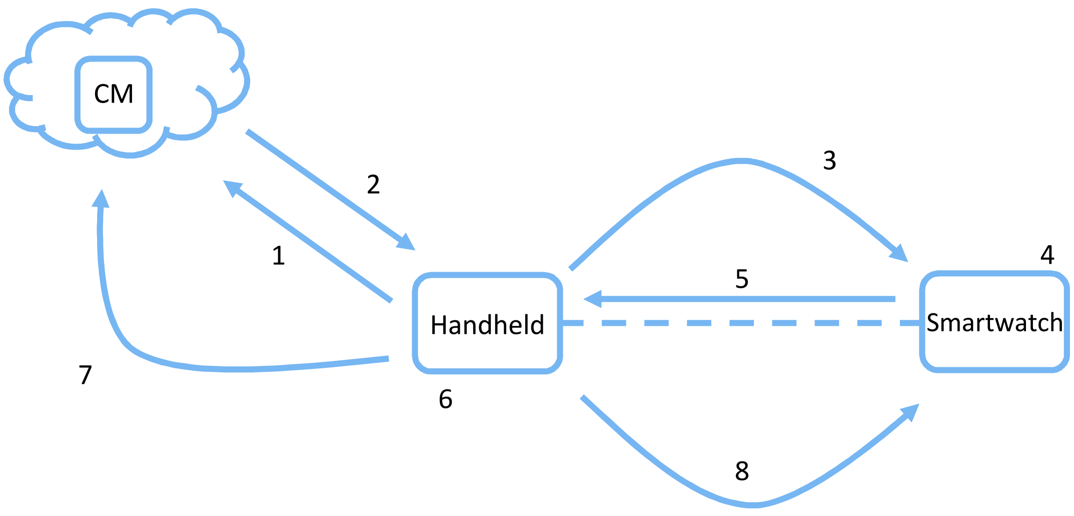


Figure 13: Smartwatch-To-Cloud Communication Protocol

Moreover, in order to receive messages from other nodes, I implemented a new Service that extends `WearableListenerService` [32], a Service offered by Android SDK that receives events from other nodes, such as data changes, messages or connectivity events. This Service is named `CiceroWearableService` (CWS), and I registered it into Android Manifest (as illustrated in Code 6). Now, every time a new message comes to a device running CBC, the method `onMessageReceived` of CWS is invoked (Step 4). This method analyzes the received message, and if is from path `TO_WEAR_MESSAGE_PATH` means that a new list of behavior is received in the wearable device. Then, CWS deserializes behaviors and start sensing them, but in a different way compared to sensing on a handheld. Indeed, CWS asks SM to start sensing, but it sets a different Situation Event listener, i.e. it instantiates a new `WearableSituationEventListener` (WSEL), instead of the regular `SituationEventListener` (SEL).

```
<application/>
    …
    <service android:name=".sentience.wear.CiceroWearableService">
        <intent-filter>
            <action
android:name="com.google.android.gms.wearable.BIND_LISTENER" />
        </intent-filter>
    </service>
</application>
```

Code 6: I registered `CiceroWearableService` in the Android Manifest

Thus, when a situation is identified, WSEL does not send directly a Situation Event to CM (anyhow it is not able to do that), but it sends the event back to the handheld. In order to do that, WSEL serializes the Situation Event instance, and sends it back to the handheld node using MessageApi again (Step 5), but with a different path (`TO_PHONE_MESSAGE_PATH`) to not be confused with previous messages.

Then, a CWS is created on the handheld device (Step 6), and the method `onMessageReceived` is invoked. But this time CWS deserializes the Situation Event in the received message, and propagate it locally as a regular Situation Event. Doing that, the regular SEL is able to send it to CM through internet (Step 7).

When the handheld needs to stop sensing also on the smartwatch, a different message is sent to the latter (Step 8). Indeed, SM sends a message to the `TO_WEAR_STOP_PATH`, and after CWS has received it, the latter stop sensing all behaviors on the smartwatch.

It is worth to note, that SCCP requires that CBC is running both on handheld, and on smartwatch. Moreover, ALP is the same library included in the project of the persuasive application for the handheld as well as in the project for the smartwatch application. Indeed, the main core of ALP is about sensing, and Classes (SOs) that SM needs for sensing. Since these Classes are needed on both kinds of devices, and since having two different libraries seems more confusing for developers, I decided to have a single project for all devices (ALP). In fact, the gain in terms of memory and battery are so small to not justify more complexity.

4.1.5. Deployment and Middleware availability

After analyzed the implementation of AEP and ALP, I am going to deal with a new problem: how distribute CBC? As choice, it was decided not to share the code as Open Source before having a solid retail version, and after a deep testing. But, at the same time, it is needed to share CBC with more developers as possible.

Regarding AEP, there is no need to share the code. Indeed, after it has been deployed to GAE platform, it starts offering CM as a service through GCE.

Regarding ALP, the problem is harder because ALP has to be made available and downloadable to developers. I decided to share ALP crating an `aar` library, after obfuscating it with ProGuard [33]. Moreover, the `aar` bundle is the binary distribution of an Android Library Project. The main difference between a `Jar` and an `aar` is that `aar`s include resources such as layouts, drawables, Android Manifest, and resources. For example, ALP needs to register some receivers into Android Manifest. Using a regular `Jar`, it is possible to share the Classes that must be registered, but not the Manifest. Then, persuasive developers have to include the registration of every receiver in persuasive application's Manifest. This is not feasible, because developers do not have to know low level details of CBC, in particular which receivers are used. Thus, using `aar` it is possible to share also Android Manifest that will be merged with the persuasive application's one.

Finally, I use ProGuard in order to obfuscate internal Classes making them harder to reverse-engineer. Indeed, only façades are not obfuscated, because persuasive applications need to access them. Thus, ALP is now completely sharable to developers, but it is a Closed Source project.

4.1.6. How developers use Cloud-Based Cicero

In this Section, I'm going to put on the developers' hat. Indeed, I will examine step by step what persuasive developers need to do in order to use CBC, and in order to implement their persuasive application.

First of all, persuasive developers need to obtained ALP, and add it as dependency to their persuasive application project. In order to do that, they have to complete following steps.

1. They have to download ALP from the Cicero's website [34], as a standalone `aar` file, or with a sample persuasive application's project [35].
2. They must add it as a new module, clicking `File > New > New Module > Import aar/jar package`, and then selecting the previously downloaded file, naming the new module `cicero-release`.
3. Then they have to add `compile project(':cicero-release')` as a new line in the app's gradle file inside dependences section.

After completing these steps, developers are able to access ALP as dependency. Moreover, they are able to invoke methods of ALP's façade named `Cicero`, but they are not easily able to access obfuscated method. Nevertheless, `Cicero` façade (CF) exposes all useful methods that developers need in order to create a persuasive application.

Then developers have to develop four Android Activities, one for each ABM Action except for Recall. Indeed, they must create Activities to be presented to users, and if it necessary notifications (Android's `Notification`) to be presented for the Recall Action. Moreover, they can develop these Activities as they prefer, for instance using a simple list of links, or presenting a more complicated user interface.

When they have created these Activities, developers have to inform CF about which Activity represents a particular ABM Action. Namely, they have to call as soon as possible (usually in the first Activity that is shown when the application is loaded) methods presented in Code 7. After doing that, CBC is ready to start sensing, and, when necessary, it is able to present to users the right Activity. When developers want to start sensing (for example after user touches a “Start” button), they have to call `connect` method of CF. Then, ALP start communicate with remote CM, asking for behaviors needed to be sensed, and then start sensing.

```java
final Cicero cicero = Cicero.getInstance();
// setting Activities
cicero.setAssessActivityAdapter(new
        AssessActivityAdapter(AssessOverviewActivity.class));
cicero.setAwareActivityAdapter(new
        AwareActivityAdapter(AwareActivity.class));
cicero.setPlanActivityAdapter(new
        PlanActivityAdapter(PlanActivity.class));
cicero.setLearnActivityAdapter(new
        LearnActivityAdapter(LearnActivity.class));
```

Code 7: Persuasive applications have to set Android Activities related to each ABM Action

It is worth to note that one of strengths of CBC is its easiness of use. Indeed, after adding ALP as a dependency, developers can have a ready-to-use persuasive application only creating four Activity, and connecting CF. After that, is up to ALP to sensing, and presenting right Activities to users. Moreover, developers do not have to set nothing about SAT in the users' application. Indeed, it is up to domain experts (in particular to DEMA) creating, or modifying behaviors in remote SAT.

Regarding DEMA, CBC provides a ready-to-use domain experts' application that lets domain experts create the SAT through an intuitive GUI. Moreover, if developers want add some functionality or shortcuts about creating SAT (for instance a preconfigured default SAT structure fitting a particular disease), they are able to change DEMA inserting this new features. In particular, developers can request adding behaviors to SAT using the related method of `Cicero` façade Class (for example `addGoalRelatedActivityPNode`), providing as parameter a new `Behavior` instance created using a factory named `BehaviorBuilder` (BB). Furthermore, BB's method `buildBehavior` lets developers easily create a behavior from its components: device, activity, context, reference value, and context and detect additional information previously described in

Section 4.1.1.3. Indeed, BB creates a new Behavior instance from provided parameters. In particular, it selects the right situation and SituationName, besides the right type for the reference value according ABM's Behavior Measurement Type (BMT). For example, from the parameters provided in Code 8, BB creates a situation with name BeginAtWithFrequency (i.e. BeginAt activity within Frequency context), and a BMT instance of type F and reference value 3. From these values besides Context and Detect additional information, BB can now create a new Behavior instance, simplifying developers' code.

```
//ACT
// setting "be at gym 3 times in a week"
Cicero.addGoalRelatedActivityPNode(
        BehaviorBuilder.buildBehavior(
                new Device(DeviceName.SmartPhone,
                           DeviceActionName.DetectUserLocation),
                new Activity(ActivityName.BeingAt),
                new Context(ContextName.Frequency),
                3,
                BehaviorUtils.stringify(
                        new FrequencyAdditionalInfo(new ContextDeadline(
                                now + 1000*60*60*24*7,
                                1000*60*60*24*7 //every week from now
                        ))),
                null)
);
```

Code 8: How developers can add a behavior to the remote SAT

It is also worth noting that developers do not have to directly invoke a remote call to a remote API in order to create the new behavior node. In fact, the remote call is hidden behind the façade Class (Cicero). In this way, all the complexity is hidden, and developers do not know something about which AsyncTask is used. Indeed, it is up to Cicero identify the right AsyncTask, and use it.

4.2. *Experimental Results*

In this section I will show the experimental results obtained testing CBC. In particular, I created a use case in which I developed several Android applications relying on CBC in order to create a realistic scenario, as shown in Figure 14. Indeed, I developed two smartphone applications, one for regular users, and one for domain experts (DEMA – see Section 4.1.3). Furthermore, I also developed a smartwatch application that can be used by users, in order to simulate a patient that owns a smartwatch in addition to a smartphone. Finally, I created an App Engine application that is tested locally, in order to simulate the cloud part of CBC where SAT resides.

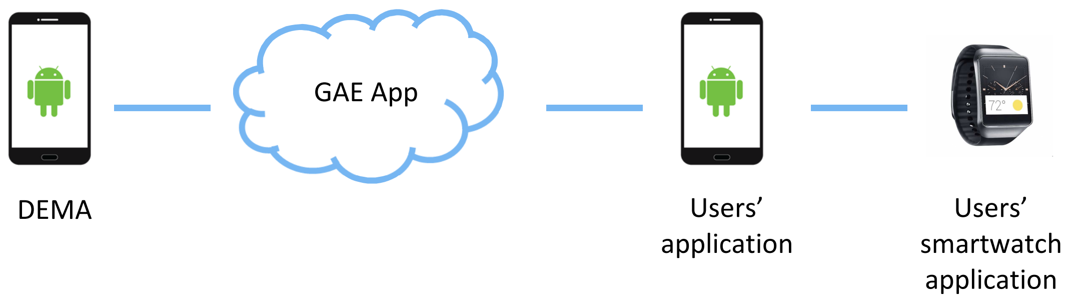


Figure 14: Use case's applications

After explaining how use case is arranged in Section 4.2.1, I am going to analyze into details the performances of the most significant applications: the users' one in Section 4.2.2, and the smartwatch one in Section 4.2.3. I have decided to not go into performance details about DEMA because the main purpose of it is present remote SAT to domain experts, and let them modify the tree. Then, performances are not the main target for this application. Regarding GAE application, the main characteristic that affects performances is the Google's implementation, and it is not the subject of this dissertation.

Finally, in Section 4.2.4 I will analyze performances of new protocol SCCP previously introduced in Sections 3.4 and 4.1.4.

4.2.1. Use case's arrangement

In order to settle the experiment, I used the arrangement shown in Figure 15. In particular, I have two physical devices connected through a USB 2.0 connection: a desktop PC, and a smartphone. Moreover, the desktop PC is a custom built system with an Intel Core i5 3570K CPU (3.4

GHz), 8GB of DDR3 RAM memory, and OSX El Capitan 10.11.2 operating system. Regarding the smartphone, I used a Vodafone VF-895N device, with the following characteristics:

- CPU: Quad-core Qualcomm Snapdragon 410 @ 1.21 GHz;
- Display: 4.97”, 720x1280 pixels, 295 dpi;
- RAM: 899 MB;
- Android Version: Android Lollipop 5.0.2.

In order to run all four applications shown in Figure 14, I installed DEMA and users’ main application on the smartphone physical device, while I emulated both GAE application and smartwatch one. The former is running locally on a GAE Development Server, but it is registered on the desktop PC’s IP address at the port 8080, so it is reachable within the local network. Besides, the latter is running on an Android emulator with the following characteristics:

- Device type: Android Wear Round;
- Display: 1.65”, 320x320 hdpi;
- RAM: 512 MB;
- VM Heap: 32 MB;
- Android Version: Android Marshmallow 6.0.

Moreover, in order to enable communication between smartphone and emulated smartwatch through USB connection, I forwarded the AVD's communication port to the connected handheld device typing the following command in the terminal: `adb -d forward tcp:5601 tcp:5601`.

Finally, both physical devices are connected to the same local network where there is a Wi-Fi router (Asus DSL-N55U). Desktop PC is connected directly to the router through a LAN connection, while smartphone is connected via Wi-Fi (IEEE 802.11n).

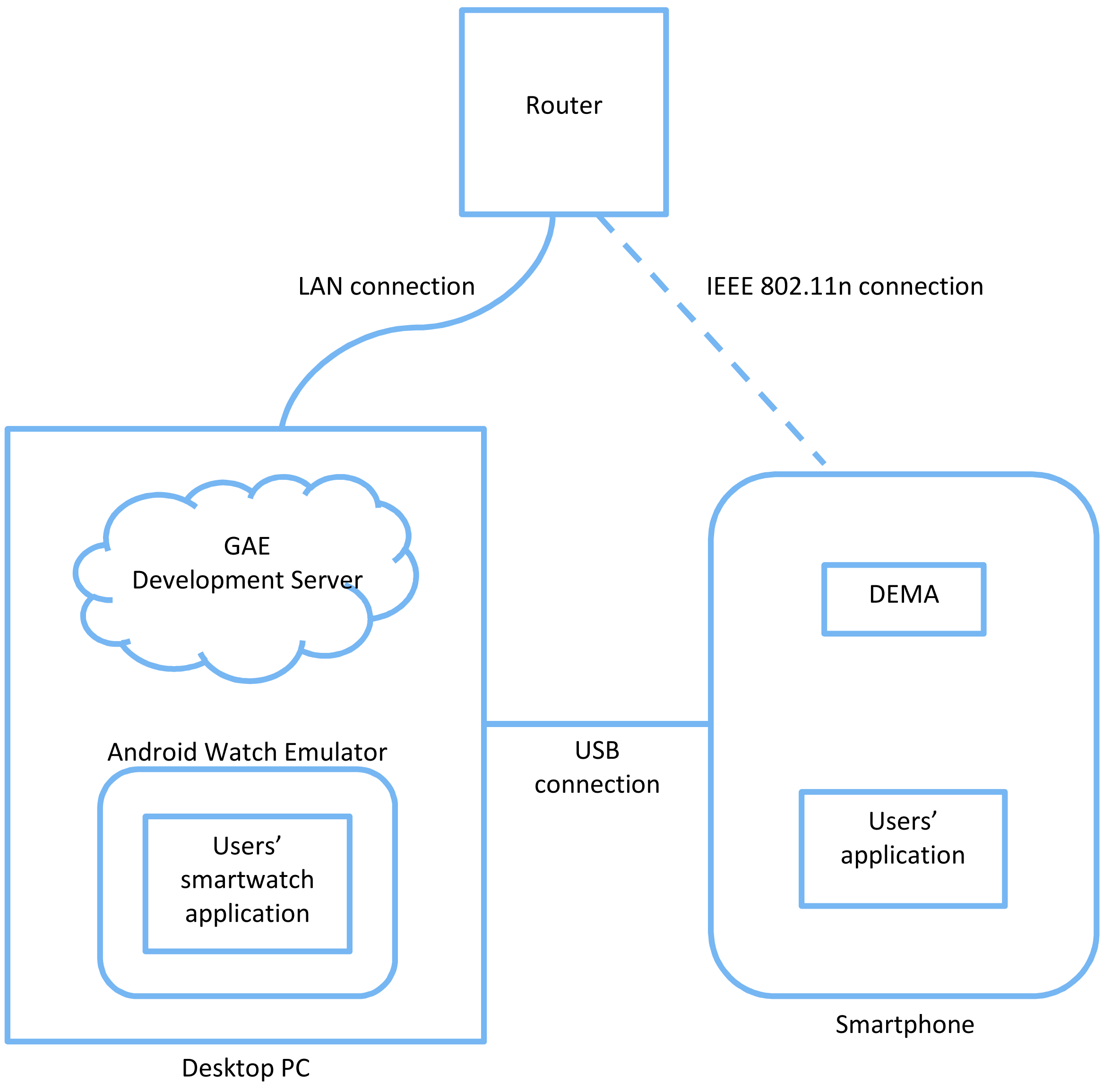


Figure 15: Use case's arrangement

Using the layout just described, I tested performance of users' applications on smartphone, and on smartwatch creating the following scenario. First of all, I launched a new GAE Development Server using Android Studio's tool, and I ran AEP on it. Secondly, I set the desktop PC's IP address as location of the server simply modifying the constant `APPENGINE_API_ROOT_URL` inside ALP's Class named `SentiencePreferences`. In this way, every persuasive application relying on ALP is able to contact AE instance. In particular, the smartphone applications, and smartwatch one are able to properly contact the GAE Development Server on the desktop PC. Thirdly, I used DEMA to create a realistic SAT. Indeed, in this use scenario I simulated a persuasive application tailored for an obese person. In this scenario a domain expert identifies behaviors shown in Table 4. Then, he or she adds each of them to SAT using DEMA, and its GUI. After the addition is completed, SAT is now ready, and a hypothetical user can start using mobile applications.

Table 4: Use case's behaviors identified by domain expert

Action Node	Sub-Action Node	P-or-N Node	Situation	Sensed on Device
Aware	Behavior-Aware	P-Node	P1: Walk more than before	Smartphone
		N-Node	N1: Walk less than before	Smartphone
Plan	Goal-Setting Status	P-Node	P2: Complete correctly goals setting	Smartphone
		N-Node	N2: Not complete goals setting	Smartphone
Learn	Behavior Knowledge	P-Node	P3: Not walk under bright light (more than 50.000 lux)	Smartphone
		N-Node	N3: Walk under bright light	Smartphone
		P-Node	P4: Walk with an heart rate less than 130 BPM	Smartwatch
		N-Node	N4: Walk with an heart rate more than 130 BPM	Smartwatch
Act	Goal-Related Activity	P-Node	P5: Be at gym at least 3 times weekly	Smartphone
		N-Node	N5: Be at gym less than 3 times weekly	Smartphone
		P-Node	P6: Walk more than 20 minutes daily	Smartphone
		N-Node	N6: Walk less than 20 minutes daily	Smartphone

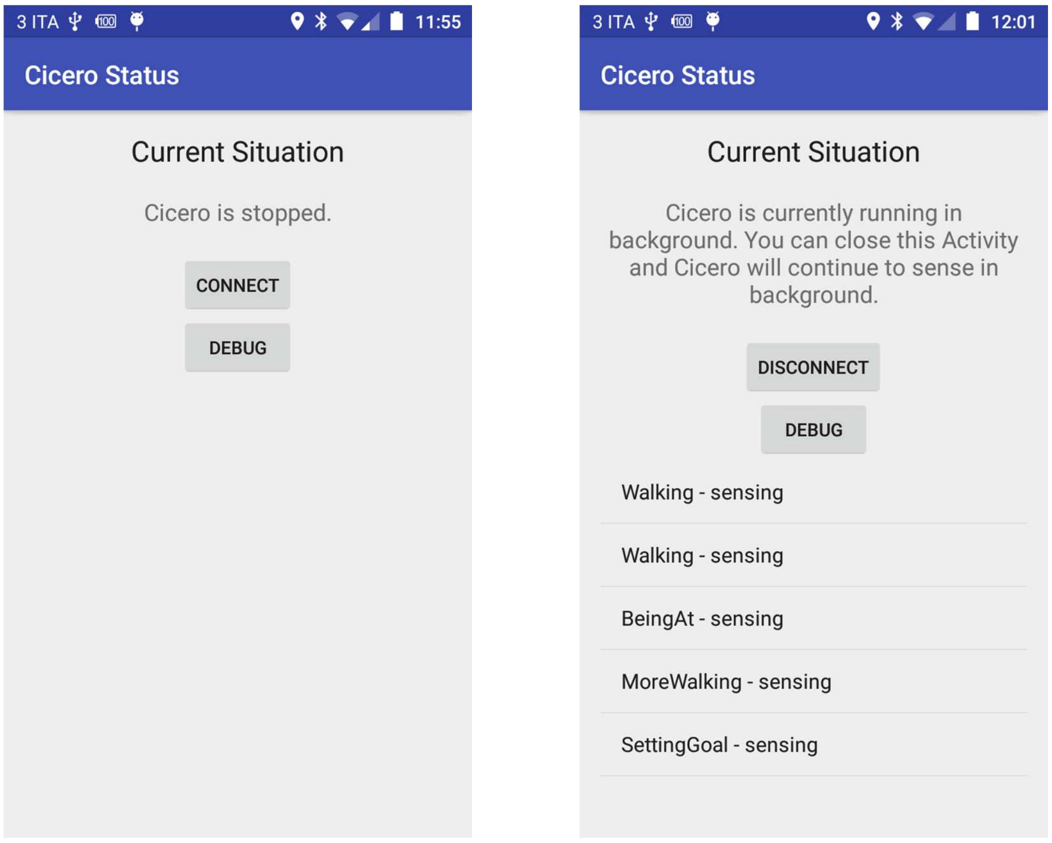
Finally, I started testing performance of users' application both on the physical smartphone, and on the emulated smartwatch. Then, I analyzed data that I will report in following sections.

4.2.2. Users' smartphone application testing

In order to test users' application installed on the smartphone, first of all I am going to show the GUI sequence that is presented to users in Subsection 4.2.2.1. Then, I will show results of performances test about most important characteristics of a mobile application: CPU (Section 4.2.2.2), memory (Section 4.2.2.3), and battery (Section 4.2.2.4) usage, besides network traffic (Section 4.2.2.5).

4.2.2.1. GUI showcase

When users open the use case's application, they can see an Activity that inform them about the status of the sensing. In particular, the first time application is opened, CBC is not sensing jet, and thus the Activity appears as the screenshot on the left in Figure 16.


3 ITA 11:55

Cicero Status

Current Situation

Cicero is stopped.

CONNECT

DEBUG

3 ITA 12:01

Cicero Status

Current Situation

Cicero is currently running in background. You can close this Activity and Cicero will continue to sense in background.

DISCONNECT

DEBUG

Walking - sensing

Walking - sensing

BeingAt - sensing

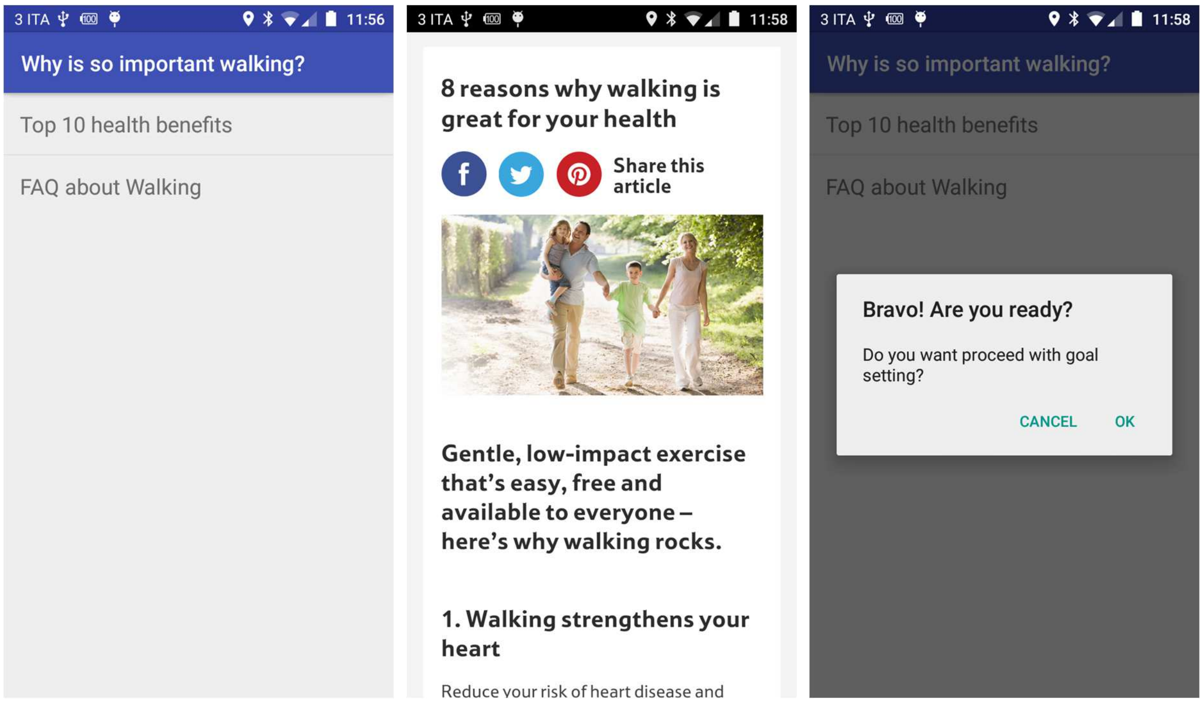
MoreWalking - sensing

SettingGoal - sensing

Figure 16: Use case's main Activity when sensing is stopped (on the left), and when is running (on the right)

Then, after starting the connection procedure tapping the button "Connect", users can navigate through a first sequence of Activities that map ABM Actions in order to complete one-time profile (e.g. gym location), and starting make users aware and teaching them. In particular, the first of these Activity is shown if Figure 17, and it make user more aware on why is so important walking. It is clear than this Activity represents the ABM's Aware Action.

Going into details, it shows to users a list of related articles, on how it is so important walking, especially for these obese patients. Users' awareness will be later assessed by nodes P1 and N1 of Table 4, sensing if they are walking more or less than before. Moreover, after a remote Assess, this is the Android Activity that will be shown if a lack of awareness will be detected. In this way, the persuasive application tries to make users more aware about their condition.


3 ITA 11:56

Why is so important walking?

Top 10 health benefits

FAQ about Walking

3 ITA 11:58

8 reasons why walking is great for your health

Share this article

Gentle, low-impact exercise that's easy, free and available to everyone – here's why walking rocks.

1. Walking strengthens your heart

Reduce your risk of heart disease and

3 ITA 11:58

Why is so important walking?

Top 10 health benefits

FAQ about Walking

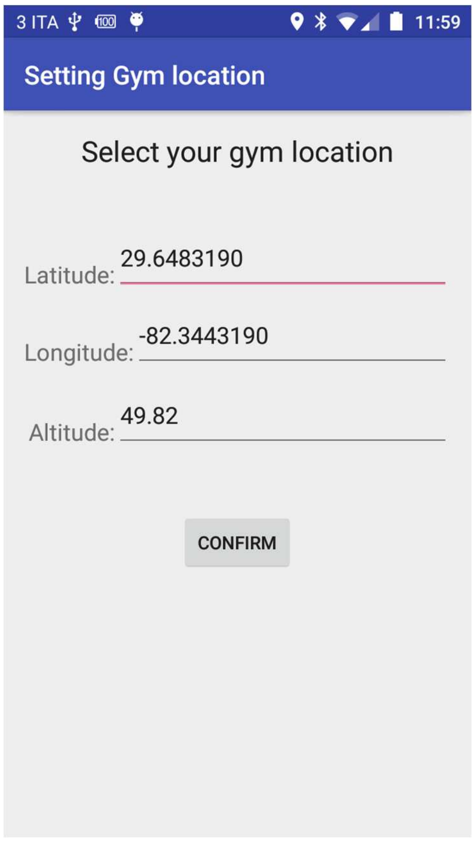
Bravo! Are you ready?

Do you want proceed with goal setting?

CANCEL OK

Figure 17: Use case's Aware Action

After finishing reading one or more contents in Aware Activity, a new Activity prompts users to insert the Gym position, in order to start sensing behavior nodes P5 and N5 in Table 4. This Activity is shown in Figure 18. Furthermore, it is worth noting that, for sake of simplicity, user needs to insert the location providing latitude, longitude, and altitude values as numbers. Nevertheless, it is up to developers to implement this Activity as they prefer, for example using Google Maps API in order to let users insert location in a simpler way (i.e. tapping on a map). Moreover, inserting gym location allows creating the ABM one-time profile shown in Figure 3, and lets ALP start sensing the related behaviors that needs a location as additional context information. Indeed, ALP needs to compare current gathered location with the defined gym location (considering inaccuracy of measurement) in order to say if user is at gym.


Setting Gym location

Select your gym location

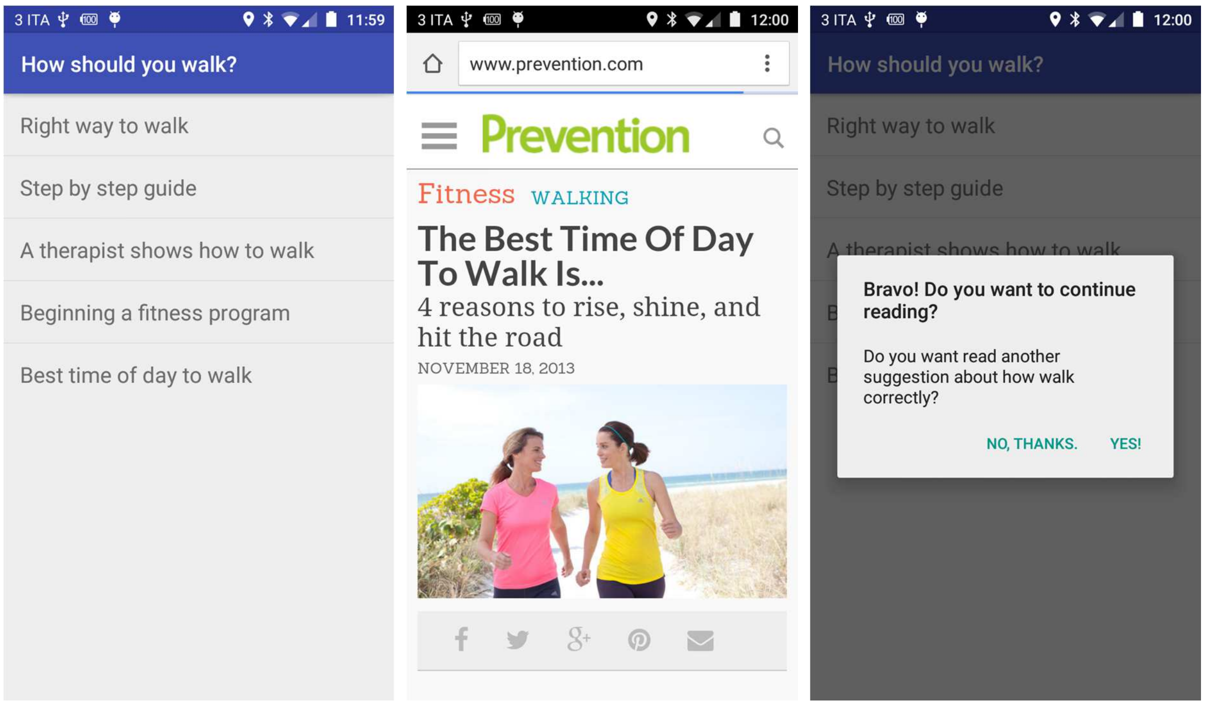
Latitude: 29.6483190

Longitude: -82.3443190

Altitude: 49.82

Figure 18: Use case's Plan Action, where users can set their gym location

Then, after setting favorite gym location, persuasive application teaches users on how walk properly. This Android Activity represents the Learn Action step in ABM. After teaching, ALP asses how users learnt sensing behaviors nodes named P3, N3, P4, and N4 in Table 4. Indeed, if a user walks on too-bright sun, or walks too fast (then heart rate increases), ALP can deduce that the user has not learnt enough because it is not recommended for obese people. Moreover, as shown in Figure 19, persuasive application presents users a list of useful links to articles or videos. Tapping on a link in the list, users can go further reading (or watching videos), and when they return back to the persuasive application, it prompts them if it is enough, or if they want read another article.



The Best Time Of Day To Walk Is...
4 reasons to rise, shine, and hit the road
NOVEMBER 18, 2013

Bravo! Do you want to continue reading?

Do you want read another suggestion about how walk correctly?

Figure 19: Use case's Learn Action

After reading all articles that he or she wants, user is guided back to the main Activity of the persuasive application (Figure 16 – right snapshot), but this time the current situation is presented. In this way users can know if CBC sensing is running, but they can also close this Activity, and ALP continue sensing in background until users will tap "Disconnect" button. It is worth noting that sensing does not require a particular GUI, because it is running in background on an Android Service. I decided to implement this main Activity in order to provide a minimum feedback to the user, but it is not mandatory for persuasive developers.

4.2.2.2. CPU usage test

In order to test CPU usage, I utilized a tool named Android Monitor [36]. This tool lets monitoring the following aspects of an application: CPU, memory, and GPU usage, besides network traffic of hardware devices.

Regarding CPU, first of all I tested the users' application installed on smartphone during its first connection (Figure 20). As previously described, during the first connection a sequence of Activity is shown. During this sequence, CPU level are higher than after sequence is completed, because of GUI creation and maintenance. Moreover, CPU reaches peaks around 15% during this first period. Later, CPU is used only for sensing, and the utilized amount of CPU is much less. It about 0.8% with some peaks around1-1.5%.

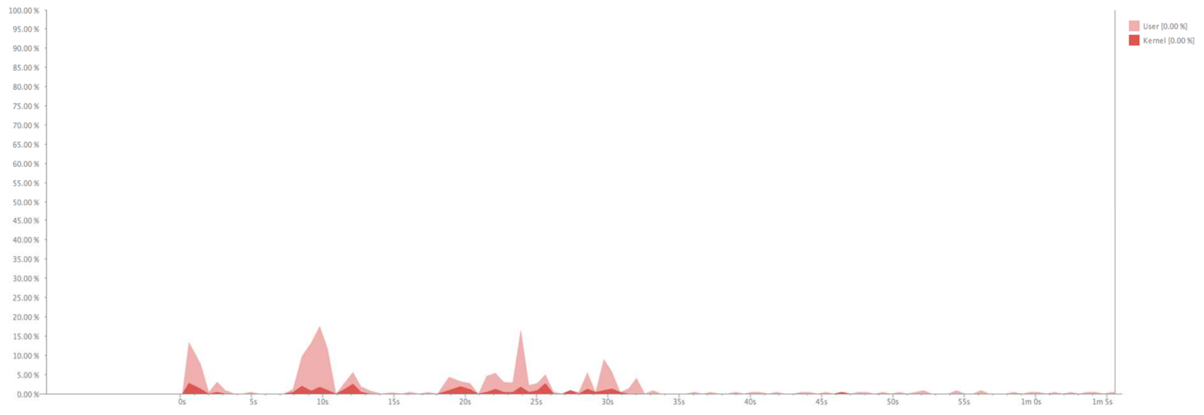


Figure 20: Mobile application's CPU usage during first connection

Then, I tested CPU in a second situation: during a later connection (Figure 21). In this case, CBC shows nothing to the user. Indeed, it has only to retrieve behaviors to sense from remote server, and then sensing them. Also in this scenario, at the beginning CPU is more used than in the second part, especially due to remote APIs calls. Once sensing is started, CPU usage is about the same that in the first scenario, i.e. around 0.8%.

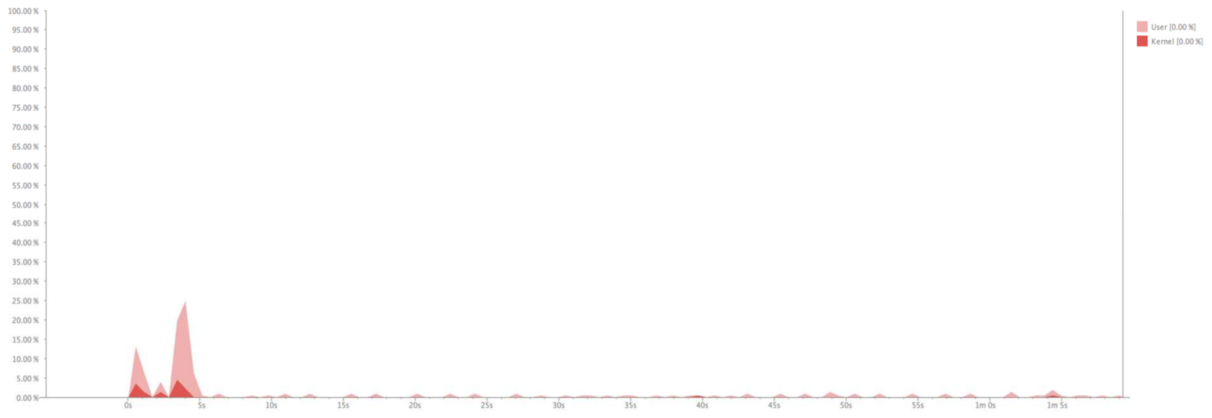


Figure 21: Mobile application's CPU usage on further connections

4.2.2.3. *Memory usage test*

As for CPU, I utilized Android Monitor also for testing memory usage. During the first connection (Figure 22), RAM usage reaches 8 MB during the sensing, and it gradually increases in the first part (i.e. while showing first cycle of Activities).

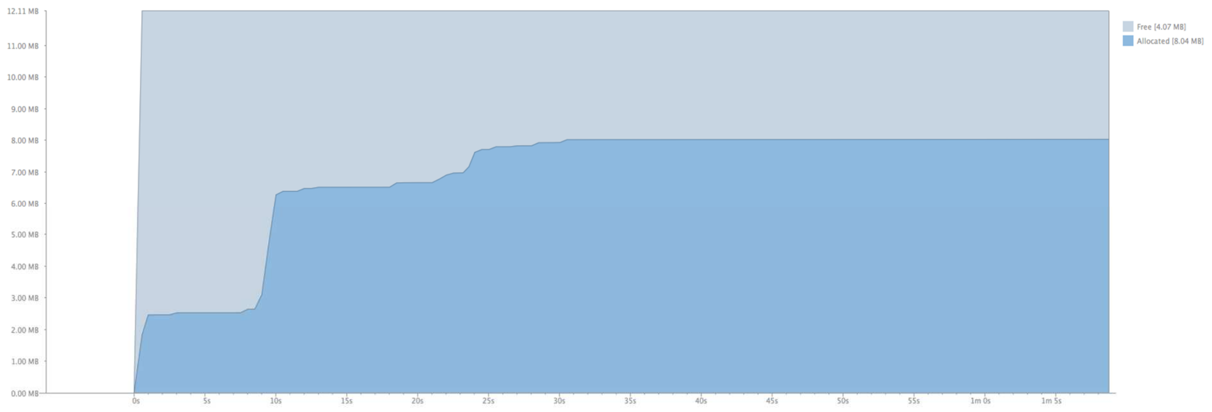


Figure 22: Mobile application's memory usage during first connection

On a further connection, memory usage grows faster than the first case, but it reaches 7 MB when sensing is started.

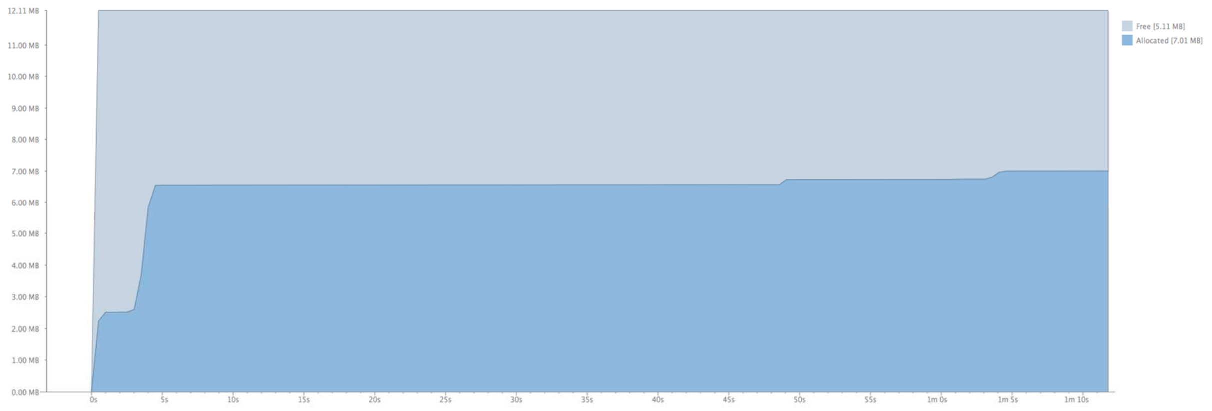


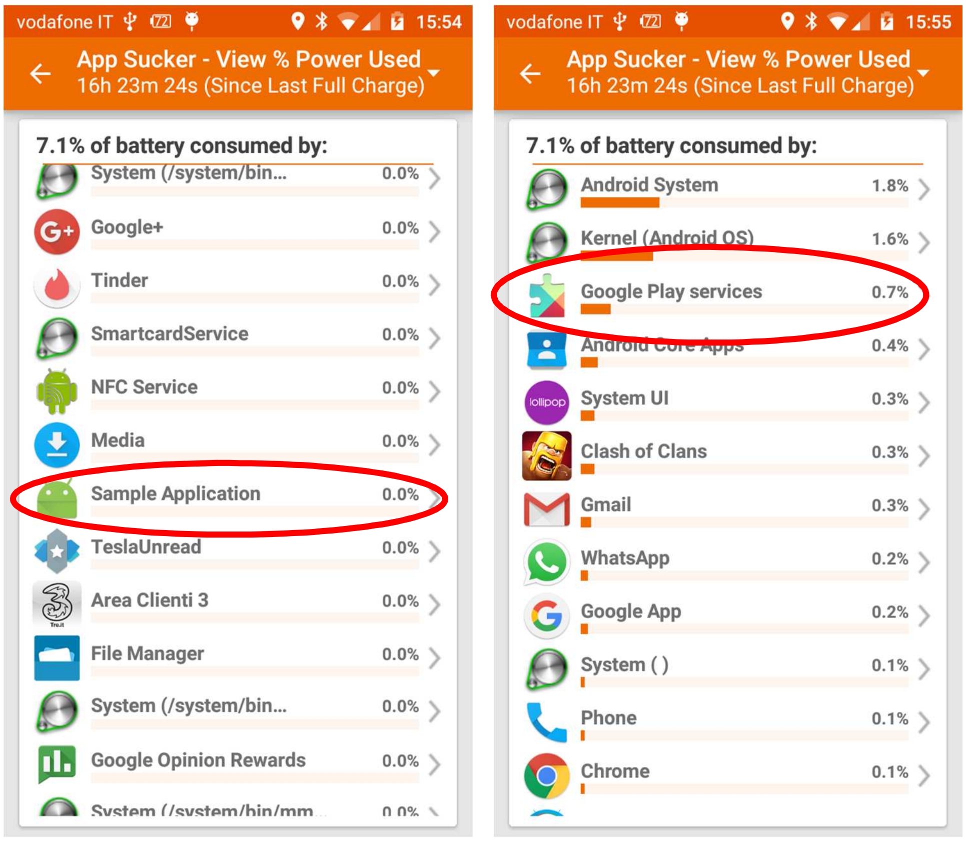
Figure 23: Mobile application's memory usage on further connections

It is worth noting that memory usage remains almost constant along all sensing. Indeed, the first effort regarding memory is spent for instantiate Sentience Objects, and storing them in the pool. Then, no more objects are instantiated, and memory usage remains almost the same.

4.2.2.4. *Battery usage test*

Nowadays, battery is a very important resource in mobile applications, and latters should use it with smartness. That said, I tested users' smartphone application also for battery usage. I used GSam Battery Monitor [37] in order to have a complete view over battery usage, in particular over which applications drain more battery. After running the use case's persuasive application on the smartphone previously described for an hour, I found that the battery consumed by persuasive application during sensing is lower than 0.1%, as shown in Figure 24.

In addition to this percentage, it is worth considering also the battery consumed by Google Play Services that is 0.7%. Indeed, CBC uses these services in order to sensing, and I must consider also them for a complete result.



vodafone IT 15:54

App Sucker - View % Power Used
16h 23m 24s (Since Last Full Charge)

7.1% of battery consumed by:

System (/system/bin...	0.0%
Google+	0.0%
Tinder	0.0%
SmartcardService	0.0%
NFC Service	0.0%
Media	0.0%
Sample Application	0.0%
TeslaUnread	0.0%
Area Clienti 3	0.0%
File Manager	0.0%
System (/system/bin...	0.0%
Google Opinion Rewards	0.0%
System (/system/bin/mm...	0.0%

vodafone IT 15:55

App Sucker - View % Power Used
16h 23m 24s (Since Last Full Charge)

7.1% of battery consumed by:

Android System	1.8%
Kernel (Android OS)	1.6%
Google Play services	0.7%
Android Core Apps	0.4%
System UI	0.3%
Clash of Clans	0.3%
Gmail	0.3%
WhatsApp	0.2%
Google App	0.2%
System ( )	0.1%
Phone	0.1%
Chrome	0.1%

Figure 24: Battery usage of use case's application on smartphone

However, the total value of battery drained in an hour by CBC in a daily use of a smartphone is lower than 1%, and it is a big plus for our middleware.

4.2.2.5. Network traffic test

Regarding network traffic, I tested with Android Monitor how much traffic is generated by the users' persuasive application installed on the smartphone.

In Figure 25, I show the graph concerning the traffic during the first connection. In this situation, most of traffic is generated in the first part, while users are loading webpages, or videos in first Activities steps. Later, network traffic is generated in order to send events to remote server, or receive downstream messages. Nevertheless, this traffic is very low: i.e. about few kilobytes per second.

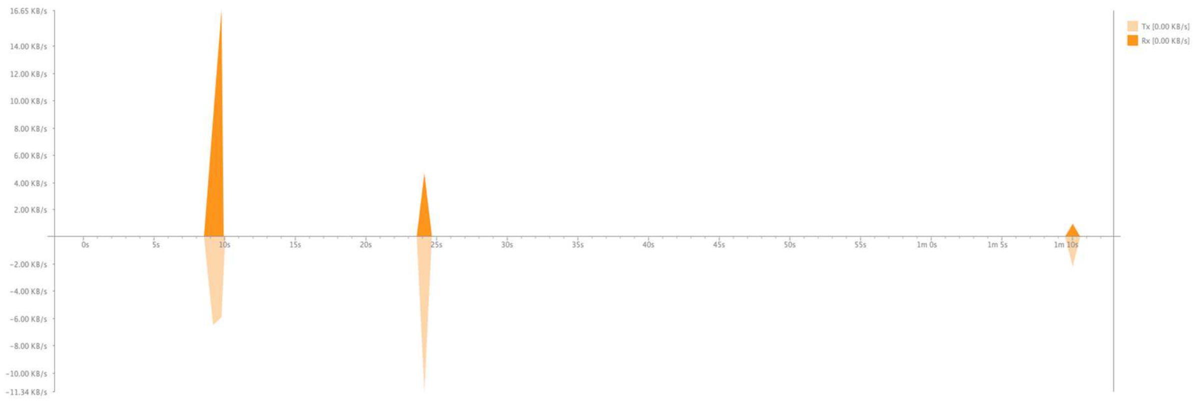


Figure 25: Mobile application's network usage during first connection

In the second scenario, which graph is shown in Figure 26, in the first part of a further connection network traffic is more that in the first connection, because client part of CBC needs to retrieve all behaviors that were previously created, and that user's smartphone needs to sense. That causes a higher traffic, anyhow around 30 KB/s for few seconds. After this first peak, the situation is the same that in the first scenario: while sensing, few packets are send in order to communicate between smartphone and server, and vice versa.

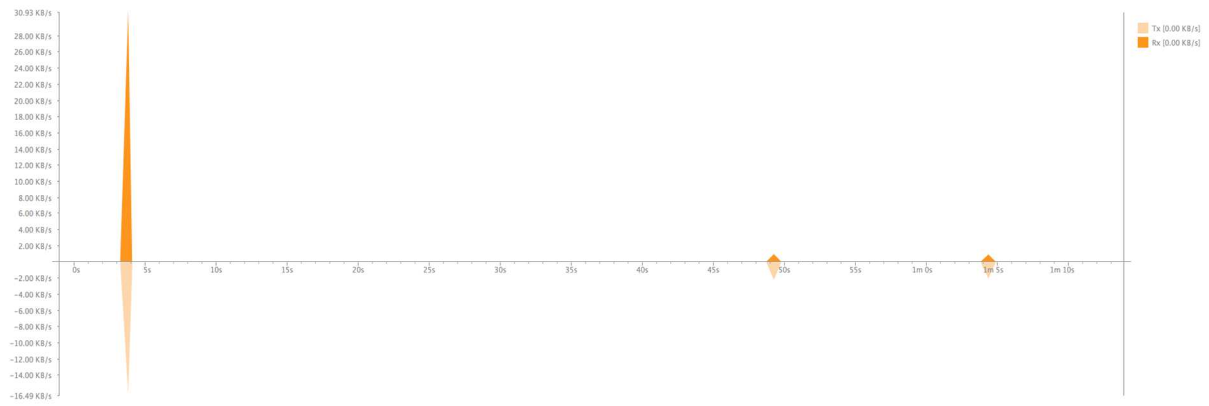


Figure 26: Mobile application's network usage on further connections

4.2.3. Smartwatch application testing

After testing smartphone application, I am going to analyze users' application installed on smartwatch. As for the smartphone one, I utilized Android Monitor in order to test some main aspect of this application.

4.2.3.1. CPU usage test

Regarding CPU usage, as shown in Figure 27 the smartwatch application does not consume a lot of resources. Indeed, for most of time CPU is not used by persuasive application, except for some isolated peaks that reach 15-20%.

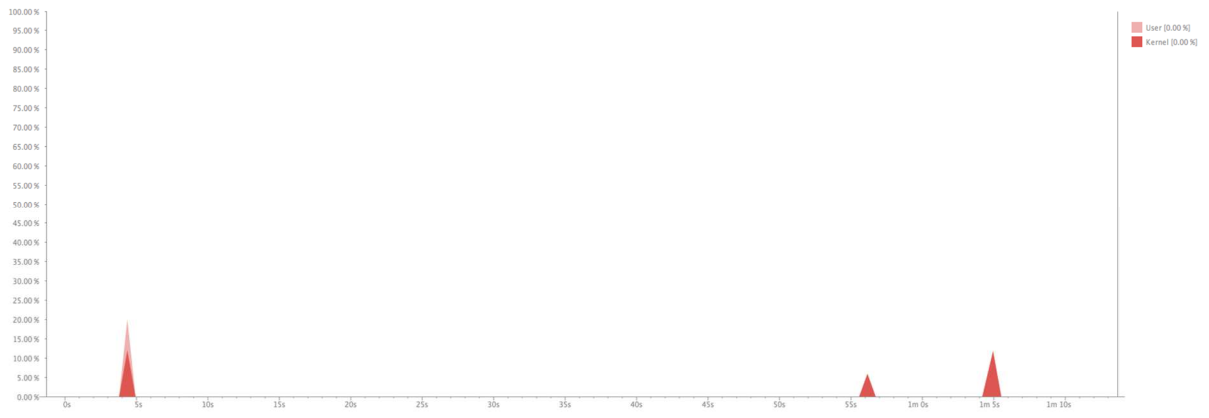


Figure 27: CPU usage in use case's application on smartwatch

However, the usage of CPU by the persuasive application is practically insignificant in the long term.

4.2.3.2. Memory usage test

Regarding memory usage, the graph of its use is shown in Figure 28, and it was obtained still using Android Monitor.

It is worth noting that memory usage in practically constant along the application life, and it is about 1.2 MB.

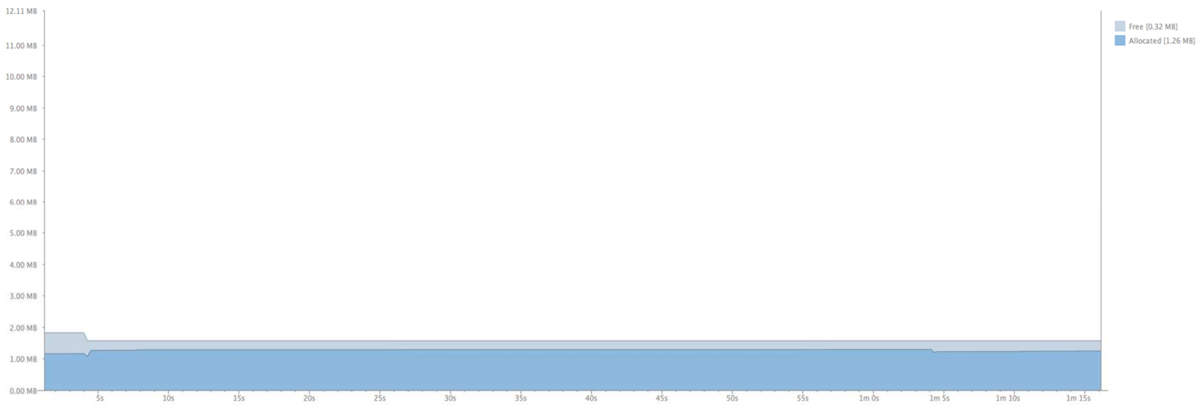


Figure 28: Memory usage in the use case's application on smartwatch

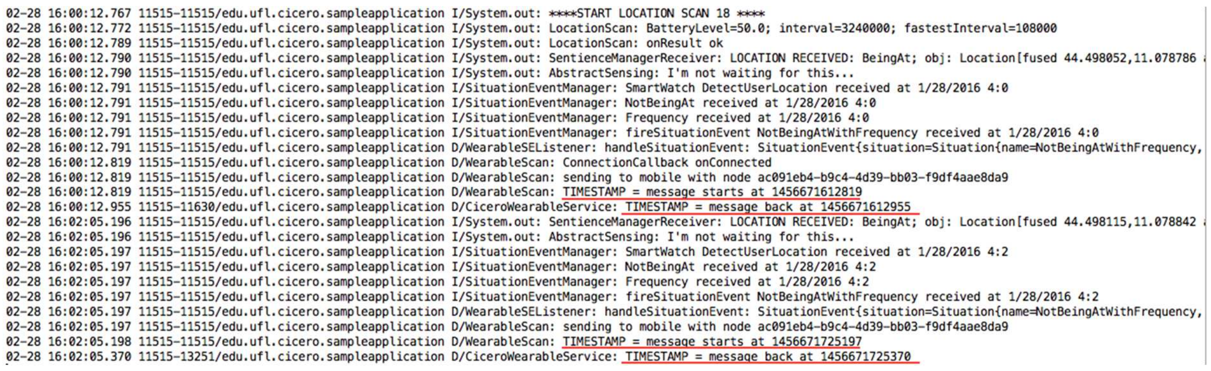
4.2.4. Smartwatch-To-Cloud Communication Protocol testing

An important improvement I made working on CBC is the creation of SCCP, previously introduced in Section 3.4, and then analyzed in Section 4.1.4. In order to conclude this analysis on performance on CBC, it is worth spending a few lines to present results about its testing.

Furthermore, in SCCP the most important latency is the one between when a situation is identified on the smartwatch, and when the message is received by the server. Indeed, this is the interval of time that characterized the CBC's implementation compared to the previous Cicero's one, because is caused by the remote position of the SAT. In other words, if SAT was local on smartphone, this latency would be smaller because there would be no remote call to GAE's APIs.

Nevertheless, in CBC model a new time needs to be considered: the time of delivery of the situation event that is propagated from ALP to AEP.

In order to test this latency, I was not able to measure it directly because of synchronization problem between clocks. Then, I decided to measure the round trip time of the message containing the situation event. Unfortunately, the protocol does not require an acknowledge message sent back to smartwatch after handheld device contact the server. However, I slightly modified it in order to send a message from handheld device back to smartwatch once the server confirmed that situation event was received. It is worth noting that this new message does not introduce a new latency, but is only needed in order to calculate the first latency I had described. Indeed, the latency between when event is generated on smartwatch, and when SAT receives the event is the half of the measured round trip time.



```
02-28 16:00:12.767 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: ****START LOCATION SCAN 18 ****
02-28 16:00:12.772 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: LocationScan: BatteryLevel=50.0; interval=3240000; fastestInterval=108000
02-28 16:00:12.789 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: LocationScan: onResult ok
02-28 16:00:12.790 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: SentienceManagerReceiver: LOCATION RECEIVED: BeingAt; obj: Location[fused 44.498052,11.078786
02-28 16:00:12.790 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: AbstractSensing: I'm not waiting for this...
02-28 16:00:12.791 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: SmartWatch DetectUserLocation received at 1/28/2016 4:0
02-28 16:00:12.791 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: NotBeingAt received at 1/28/2016 4:0
02-28 16:00:12.791 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: Frequency received at 1/28/2016 4:0
02-28 16:00:12.791 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: fireSituationEvent NotBeingAtWithFrequency received at 1/28/2016 4:0
02-28 16:00:12.791 11515-11515/edu.ufl.cicero.sampleapplication D/WearableSEListener: handleSituationEvent: SituationEvent{situation=Situation{name=NotBeingAtWithFrequency,
02-28 16:00:12.819 11515-11515/edu.ufl.cicero.sampleapplication D/WearableScan: ConnectionCallback onConnected
02-28 16:00:12.819 11515-11515/edu.ufl.cicero.sampleapplication D/WearableScan: sending to mobile with node ac091eb4-b9c4-4d39-bb03-f9df4aae8da9
02-28 16:00:12.819 11515-11515/edu.ufl.cicero.sampleapplication D/WearableScan: TIMESTAMP = message starts at 1456671612819
02-28 16:00:12.955 11515-11630/edu.ufl.cicero.sampleapplication D/CiceroWearableService: TIMESTAMP = message back at 1456671612955
02-28 16:02:05.196 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: SentienceManagerReceiver: LOCATION RECEIVED: BeingAt; obj: Location[fused 44.498115,11.078842
02-28 16:02:05.196 11515-11515/edu.ufl.cicero.sampleapplication I/System.out: AbstractSensing: I'm not waiting for this...
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: SmartWatch DetectUserLocation received at 1/28/2016 4:2
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: NotBeingAt received at 1/28/2016 4:2
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: Frequency received at 1/28/2016 4:2
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication I/SituationEventManager: fireSituationEvent NotBeingAtWithFrequency received at 1/28/2016 4:2
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication D/WearableSEListener: handleSituationEvent: SituationEvent{situation=Situation{name=NotBeingAtWithFrequency,
02-28 16:02:05.197 11515-11515/edu.ufl.cicero.sampleapplication D/WearableScan: sending to mobile with node ac091eb4-b9c4-4d39-bb03-f9df4aae8da9
02-28 16:02:05.198 11515-11515/edu.ufl.cicero.sampleapplication D/WearableScan: TIMESTAMP = message starts at 1456671725197
02-28 16:02:05.370 11515-13251/edu.ufl.cicero.sampleapplication D/CiceroWearableService: TIMESTAMP = message back at 1456671725370
```

Figure 29: Log message with some timestamp used for calculating the round trip time

After calculating several round trip times, the average time in the test arrangement is around 170 milliseconds, and then the latency between smartwatch and SAT is around 85 milliseconds, with a maximum value of 133 milliseconds and a minimum value of 52 milliseconds. As an example, in Figure 29 some timestamps used to calculate the round trip time are underlined. In these two cases, round trip time was 136 and 172 milliseconds.

4.2.5. Considerations about experimental results

As a conclusion of this section about experimental results, it is worth noting that CBC results to be a very light middleware. Indeed, CPU usage while sensing has peaks of only 0.8%, memory usage is under 7MB, and battery consumption is extremely low (i.e. more than 1% for an hour of sensing considering also Google Play Services drain). Also smartwatch application recorded very good results, both about CPU and memory usage. Furthermore, SCCP has a low latency in the test arrangement, where server is on local network.

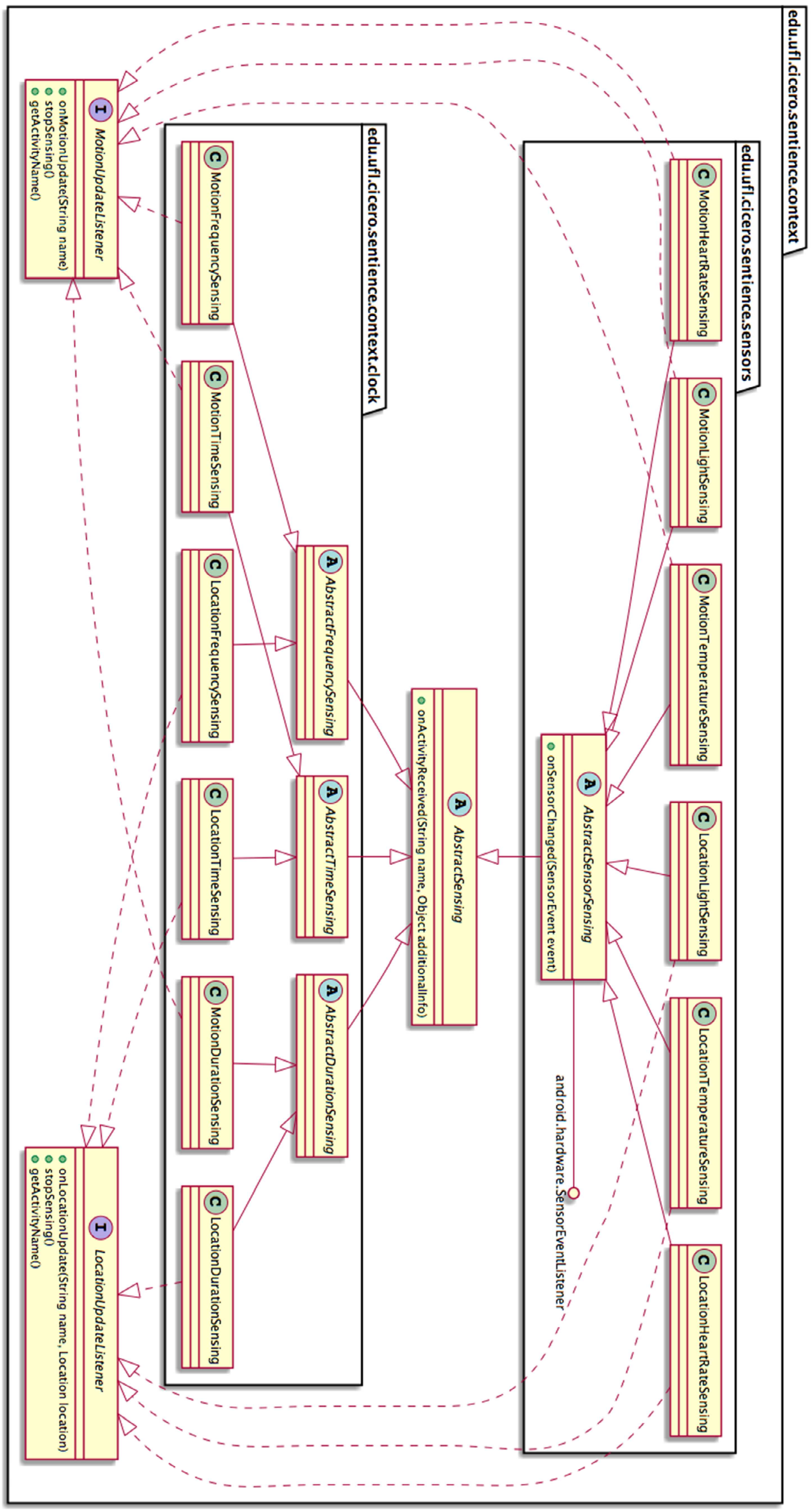
That lightness is a great feature supporting CBC as middleware for persuasive application.

Appendix A - Code snippets

A.1. GetTreeSituationAsyncTask

```
public class GetTreeSituationAsyncTask extends AsyncTask<Void, Void,
  StringResponse> {
      public static final String TREE_SITUATION_KEY = "cicero_t_s";
      private Context context;
      private Class<? extends Activity> secondClass;
      private final ProgressDialog pd;

      public GetTreeSituationAsyncTask(Context context, Class<?
  extends Activity> secondClass) {
          this.context = context;
          this.secondClass = secondClass;
          pd = ProgressDialog.show(context, "Loading", "Wait while
  loading...");
      }

      @Override
      protected StringResponse doInBackground(Void... params) {
          try {
              StringResponse sr =
  Cicero.getDomainExpertApiService().tree().execute();
              return sr;
          } catch (IOException e) {
              e.printStackTrace();
          }
          return null;
      }

      @Override
      protected void onPostExecute(StringResponse stringResponse) {
          if (stringResponse==null) {
              Toast.makeText(context, "Error
  connecting",Toast.LENGTH_LONG).show();
          } else {
              Logger.getLogger("Tree: ").log(Level.INFO, "Tree: " +
  stringResponse.getString());

              Bundle bundleObject = new Bundle();
              Intent i = new Intent(context, secondClass);
              bundleObject.putString(TREE_SITUATION_KEY,
  stringResponse.getString());
              i.putExtras(bundleObject);
              context.startActivity(i);
          }
          if (pd != null)
              pd.dismiss();
      }
}
```

A.2. SO Hierarchy



Conclusion

Nowadays, persuasion is pervasive and is part of our lives. Every day, we are affected consciously or unconsciously by persuasive messages especially from mass media, and they try to change our behaviors. Furthermore, also computers and other smart devices are becoming more and more persuasive. Indeed, they are assuming roles that were traditionally filled by specialists, speakers, or salespeople. The research area that covers these new persuasive media and modalities is named captology, thus the study of interactive computing systems designed to change people's attitudes and behaviors.

Despite the increasing success of captology, there seems to be a lack of frameworks that help developers to build mobile persuasive applications. This gap was filled by Cicero, an Android middleware aimed to support developers in their efforts. Indeed, Cicero simplifies amount of lines of code that developers have to write, and it eases the theoretical knowledge needed by them to develop a persuasive app. In fact, developers can focus themselves on business intelligence of their application, letting Cicero handle persuasive effort basing on ABM, and SAT.

This dissertation has presented captology, and its advantages, focusing on how a computer, or an application can be considered a real-effective persuader. Then, I analyzed both the theoretical middleware and Cicero. In particular, I went into details of ABM, its components, and SAT, a methodology and an algorithm for domain-specific behavior assessment under ABM. Besides these two models, I detailed Cicero, both its model and implementation.

I also presented my enhancements to Cicero. Moreover, I paid particular attention to aspects that can be improved. In particular, I analyzed how Cicero is mono sensing, and mono device. Then, I detailed my solutions to these problems, namely a new sensing architecture that allows multi sensing, the new cloud-based CBC model offering assess as a service, and the new SCCP (i.e. a protocol to include Smartwatches in devices that can be used in Cicero’s sensing).

Then, I initially presented the server side implementation, included the technologies used, among which Google App Engine, Google Cloud Endpoints, and Google Cloud Messaging. After that, I analyzed client side implementation, showing the Sentience Pool with its Sentience Objects, and the bridge between smartwatches and the cloud in order to enable communications between these components.

I also arranged a use case of CBC, testing its performances. It resulted that CPU, RAM, and battery usages, besides generated network traffic, are good thanks to new simple sensing model based on optimized Google Play Services. Moreover, the client part of CBC running on smartphone or smartwatches is now lacking of the Cyber Manager module (currently on cloud), implying less consumed resources on users’ devices. Hence, this good results are one of the main strengths of CBC, besides its simplicity of use that allows developers to build a persuasive application writing very few Android Activities, and nothing more.

To summarize, I created a new version of Cicero based on a cloud architecture. Furthermore, the former is now multi sensing, and can be developed in a multi-device scenario. Moreover, developers are now able to create applications also for Smartwatches, thanks to a new protocol I created to let cloud and Smartwatches communicate. In addition, performances are very good for a sensing middleware, proving that Cicero consumes a very little amount of CPU, memory, and battery. That, besides its simplicity of use, makes Cicero a very good choice for persuasive developers.

Finally, some possible future extensions could be improvements to the cloud part of Cicero. In particular, some aspects that can be enhanced are the security of the SAT, contemplating a users’ registration system in order to better protect sensible information, and the management of a multi-users system, in which several users and several domain experts are able to act concurrently.

Bibliography

1] B. J. Fogg, "Persuasive computers: perspectives and research directions," in *SIGCHI Conference on Human Factors in Computing Systems*, New York, 1998, pp. 225-232.

2] B. J. Fogg, *Persuasive Technology: Using Computers to Change What We Think and Do (Interactive Technologies)*, 1st ed.: Morgan Kaufmann, 2002.

3] B.J. Fogg, Cuellar G., and D. Danielson, "Motivating, Influencing, and Persuading Users," in *Human-Computer Interaction Fundamentals*.: CRC Press, 2009, pp. 133-147.

4] D. Lee, S. Helal, and B. Johnson, "An Action-based Behavior Model for Persuasive Telehealth.," in *8th International Conference on Smart Homes and Health Telematics (ICOST)*, Seoul, S. Korea, 2010.

5] D. Lee, Models and Theory for Pesuasion-Aware Computing, 2012, PhD Dissertation.

6] D. Lee, S. Helal, Y. Sung, and S. Anton, "Situation-Based Assess Tree for User Behavior Assessment in Persuasive Telehealth," *Human-Machine Systems*, vol. 45, no. 5, pp. 624-634, October 2015.

7] A. D'Aloia, Persuasive Computing: an Android-based Framework for Developing and Managing Persuasive Applications, 2015, Master Degree's dissertation at the University of Bologna.

8] Android Sensors Overview. [Online]. http://developer.android.com/guide/topics/sensors/sensors_overview.html

9] ActivityRecognitionApi. [Online]. https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi

10] FusedLocationProviderApi. [Online]. https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi

11] SensorManager. [Online]. http://developer.android.com/reference/android/hardware/SensorManager.html

12] TelephonyManager. [Online]. http://developer.android.com/reference/android/telephony/TelephonyManager.html

13] MediaRecorder. [Online]. http://developer.android.com/reference/android/media/MediaRecorder.html

14] D. Lee and S. Helal, "From Activity Recognition to Situation Recognition," in *11th International Conference on Smart Homes and Health Telematics (ICOST)*, Singapore, 2013.

15] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones," in *Hot-Mobile*, 2012.

16] T. Kaler et al., "Code In The Air: Simplifying Sensing on Smartphones," in *SenSys*, 2010.

17] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland, "Social fMRI: Investigating and shaping social mechanisms in the real world," in *Pervasive and Mobile Computing*, 2011.

[18] Tasker. [Online]. http://tasker.dinglisch.net/

[19] P. Vaka, F. Shen, M. Chandrashekar, and Y. Lee, "PEMAR: A Pervasive Middleware for Activity Recognition with Smart Phones," in *PerCom Workshops*, 2015.

[20] H. Mukhtar, A. Ali, D. Belaid, and S. Lee, "Persuasive Healthcare Self-Management in Intelligent Environments," in *International Conference on Intelligent Environments (IE)*, 2012.

[21] G. Marcu, J.E. Bardram, and S. Gabrielli, "A framework for overcoming challenges in designing persuasive monitoring and feedback systems for mental illness," in *PervasiveHealth*, 2011.

[22] Google Fit SDK. [Online]. https://developers.google.com/fit/

[23] Google Play services location APIs. [Online]. https://developers.google.com/android/reference/

[24] G. Cardone, A. Cirri, A. Corradi, L. Foschini, and R. Montanari, "Activity recognition for Smart City scenarios: Google Play Services vs. MoST facilities," in *Computers and Communication (ISCC)*, 2014.

[25] A. Zhan, J.H. Lim, and A. Terzis, "DailyAlert: A Generic Mobile Persuasion Toolkit for Smartphones," in *PhoneSense*, 2011.

[26] Google App Engine. [Online]. https://cloud.google.com/appengine/

[27] Google Cloud Endpoints. [Online]. https://cloud.google.com/appengine/docs/java/endpoints/

[28] Google Cloud Messaging. [Online]. https://developers.google.com/cloud-messaging/

[29] Google GCM API. [Online]. https://developers.google.com/android/reference/com/google/android/gms/gcm/package-summary

[30] Objectify. [Online]. https://github.com/objectify/objectify

[31] Google MessageApi. [Online]. https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi

[32] WearableListenerService. [Online]. https://developers.google.com/android/reference/com/google/android/gms/wearable/WearableListenerService

[33] ProGuard. [Online]. http://proguard.sourceforge.net/

[34] Cicero's website. [Online]. http://www.icta.ufl.edu/cicero/

Sample persuasive application developed using CBC. [Online].

35] https://github.com/persim/cicero-sample/

Android Monitor. [Online]. http://developer.android.com/tools/help/android-monitor.html

36]

GSam Battery Monitor. [Online].

37] https://play.google.com/store/apps/details?id=com.gsamlabs.bbm&hl=en

Acknowledgements

First of all, I would like to express my heartfelt gratitude to the Professor Paolo Bellavista for giving me the opportunity of make such incredible experience, both as a student and as a man, and for the help he always provided me.

I would also like to thanks Professor “Sumi” Helal for the challenging project that offered to me, for the help he provided me about working and living in Gainesville, and for making me feel like at Bologna in his laboratory.

I am grateful to Antonello D’Aloia for the incredible help he gave me understanding Cicero and his previous work at the Persuasive Laboratory, for the long Skype calls, and for all efforts we made on the paper we published about Cicero.

I cannot mention my family, that always support me in all my study career, and especially in this experience away from home. I will never forget your love and what you made for me, in particular from who is not here to see me.

Last but not least, I would like to thanks all my friends that supported me, both here in Italy (Lorenzo, Riccardo, Richard inter alia) and in Gainesville (Quincy – I really appreciated your help and time we spent together to see new places – and Ahmed – who always help me with a smile). I am really grateful to all of you.