Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, INFORMATICA E DELLE TELECOMUNICAZIONI

Ciclo XXIV

Settore Concorsuale di afferenza: 09/H1

Settore Scientifico disciplinare: ING-INF/05

Middleware for quality-based context distribution in mobile systems

Presentata da: Mario Fanelli

Coordinatore Dottorato — Chiar.mo Prof. Ing. Luca Benini

Relatore — Chiar.mo Prof. Ing. Antonio Corradi

Esame finale anno 2012

TABLE OF CONTENTS

Abstract

The continuous advancements and enhancements of wireless systems are enabling new compelling scenarios where mobile services can adapt according to the current execution context, represented by the computational resources available at the local device, current physical location, people in physical proximity, and so forth. Such services called context-aware require the timely delivery of all relevant information describing the current context, and that introduces several unsolved complexities, spanning from low-level context data transmission up to context data storage and replication into the mobile system. In addition, to ensure correct and scalable context provisioning, it is crucial to integrate and interoperate with different wireless technologies (WiFi, Bluetooth, etc.) and modes (infrastructure-based and ad-hoc), and to use decentralized solutions to store and replicate context data on mobile devices. These challenges call for novel middleware solutions, here called Context Data Distribution Infrastructures (CDDIs), capable of delivering relevant context data to mobile devices, while hiding all the issues introduced by data distribution in heterogeneous and large-scale mobile settings. This dissertation thoroughly analyzes CDDIs for mobile systems, with the main goal of achieving a holistic approach to the design of such type of middleware solutions. We discuss the main functions needed by context data distribution in large mobile systems, and we claim the precise definition and clean respect of quality-based contracts between context consumers and CDDI to reconfigure main middleware components at runtime. We present the design and the implementation of our proposals, both in simulation-based and in real-world scenarios, along with an extensive evaluation that confirms the technical soundness of proposed CDDI solutions. Finally, we consider three highly heterogeneous scenarios, namely disaster areas, smart campuses, and smart cities, to better remark the wide technical validity of our analysis and solutions under different network deployments and quality constraints.

Keywords: Mobile Systems, Context Awareness, Context Data Distribution Infrastructure, Quality of Context.

1. Introduction

The widespread adoption of wireless technologies and mobile devices is pushing toward the provisioning of Internet-based services in an 'anytime and anywhere' manner. Mobile users assume to be able to access their own full service set while freely roaming between different physical places, and potentially changing the wireless technology used to enable service provisioning. Among other typical mobile services, in the last two decades we witnessed the uprising of a new class of services, usually called context-aware, capable of adapting their own runtime behaviour according to current working conditions, such as computational capabilities of the mobile device, people co-located in the physical surroundings, and so forth [1, 2].

Although both the capacity of the wireless networks and the computational power of mobile devices are constantly growing, the development of context-aware services in large-scale mobile settings is still an extremely complex and challenging task. From the network viewpoint, the provisioning of context information introduces a management traffic strictly related with the number of roaming mobile devices. In addition, since device mobility can implicitly lead to frequent context changes, context-aware services potentially require a continuous delivery of context data to promptly trigger adaptations based on up-to-date and precise context information. At the same time, both the storage and the processing of large amounts of context data produced into the mobile system require extremely distributed and scalable architectures capable of efficiently dealing with CPU and memory constraints, especially for more limited mobile devices. Finally, such challenging issues become even more complex when we consider the introduction and the enforcement of agreed quality levels on context provisioning; above all, the emerging notion of Quality of Context (QoC) enables context producers and consumers to negotiate the quality of provisioned context data, as well as of the involved delivery process [3, 4].

To be more clear, in our opinion context provisioning in large-scale mobile systems has to deal with main issues that we group along three directions:

- *Heterogeneity and mobile device resource limitations* - Future mobile systems feature extremely heterogeneous mobile devices, that are battery-powered and with tight CPU and memory constraints. Apart from local limitations, mobile devices can exploit heterogeneous wireless infrastructures with scarce bandwidth, that can also change according to the employed technology. In addition, even if modern devices

are equipped with several wireless interfaces, communication opportunities have to be carefully managed to prevent excessive battery consumption.

- *Context data management and delivery scalability* - Large-scale mobile systems feature thousands of sensors that continuously produce new context data, with different expressiveness, data payload, and production rates, all strictly related with described context aspects. Sensors can be deployed either on mobile devices or on wireless fixed infrastructures; although the former case is appealing due to the direct provisioning of context, it leads to higher device costs. As the system scale grows, both context data storage and processing introduce a fundamental scalability bottleneck; similarly, context data distribution, both from fixed infrastructures to mobile devices, and vice versa, has to be properly tailored depending on available resources.
- *Quality-based context provisioning* - Quality constraints, both on the context data and on the distribution process, have a fundamental role in order to prevent useless and noisy adaptations. Real-world sensors introduce errors due to physical limitations, and multiple context producers can create and inject conflicting context data into the system. Similarly, unreliable wireless infrastructures can result in partial and imprecise context provisioning due to both transmission delays and packet droppings.

To address the above issues, several framework and middleware solutions have been proposed in the research literature. However, to the best of our knowledge, previous works primarily focused on local issues, such as efficient context representation and notifications to running services, while typically leaving out the great deal of complexity introduced by the provisioning of useful context data to roaming mobile devices. Hence, although most previous efforts presented interesting and valid solutions for the sake of local context provisioning, this thesis work is motivated by the fact that additional research is needed to enable scalable and quality-based context data delivery in large-scale mobile systems.

Our dissertation addresses this lack by highlighting the main challenges introduced by context data delivery in mobile systems, and by proposing new architectural models, as well as design choices, for Context Data Distribution Infrastructures (CDDIs) with scalability and quality goals. One of the main thesis claims is that CDDIs have to opportunistically exploit any mobile device and all connectivity opportunities to reduce context data distribution overhead, while enforcing quality constraints in context

provisioning. With finer degree of details, CDDIs should adopt both heterogeneous wireless standards, e.g., IEEE 802.11 (WiFi) and Bluetooth (BT), and modes, i.e., infrastructure-based and ad-hoc, to increase system scalability at the communication layer. If correctly handled, the usage of mobile devices as context data carriers can greatly reduce runtime data traffic over more limited and expensive fixed wireless infrastructures, by granting mobile devices exchange data through ad-hoc wireless links. This also alleviates the context data storage burden as the CDDI can exploit the memorization resources available on the mobile devices. In addition to this major claim, and to make a very synthetic preposition, the manifold contributions of this thesis can be grouped in the following main areas:

- *The introduction of a new unifying logical model for CDDIs in large-scale mobile systems* - The thesis proposes a high-level logical architecture, by detailing main modules with associated realization issues and design choices. We thoroughly evaluate different realization choices, while also analyzing the impact of network deployments on context data distribution functions. To better assess the validity of our logical model, we used it to classify the most important research works currently available in literature [5].
- *The design and the implementation of different CDDIs, targeted for three different and significant case studies* - By considering that network deployments greatly affect the realization of context provisioning mechanisms, we organize part of this dissertation along three main case studies: *context-aware services for emergency response scenarios*, *context-aware services for smart university campuses,* and *context-aware services for smart cities.* In this way, we aim to better highlight how the main CDDI mechanisms interact and are affected by network deployment. As the system scale grows, we will introduce and employ different context data distribution protocols and solutions, in order to keep on ensuring system scalability and quality-based constraints.
- *The implementation of the main CDDIs components*, *both on network simulators and on real testbeds* - To assess the technical soundness of our proposals, we implemented the main mechanisms introduced by our CDDIs as software components. In particular, we provide 1) a simulator-based implementation, useful to validate distribution protocols in large-scale systems and with different mobility patterns; and 2) a real-world implementation, useful to test important performance

indicators, such as CPU and memory overhead, on real deployments. By jointly exploiting such implementations, we aim to better evaluate our solutions in large-scale systems, while always considering the runtime overhead that such solutions would introduce in real deployments.

- *A thorough evaluation of the performance reached by proposed context data distribution protocols, based on both simulations and on-the-field prototyping* - For each case study, we extensively evaluate the effectiveness and the efficiency of proposed context data distribution mechanisms. If useful, such evaluation will be carried out by exploiting both approaches of simulation-based and real testbeds results.

The thesis is organized along eight main chapters divided in two main parts. Starting with the first part, Chapter 2 introduces useful background knowledge, by detailing main context definitions and categorizations, as well as quality constraints in context data delivery. Chapter 3 presents and discusses both the main issues and the design guidelines for CDDIs in mobile systems, and introduces important related works in order to better focus main gaps in the current research literature. Finally, Chapter 4 introduces our novel CDDI logical model, by also discussing main involved layers, possible design choices, and network deployments; in addition, it presents important differences with traditional data distribution mechanisms available in literature, so to better highlight the need of additional research in this area.

After this first part, primarily focused on modeling CDDI requirements and providing usable and valid design guidelines, Chapter 5 enacts as glue between the two thesis parts, and details our main case studies, in order to clarify both the main issues introduced by adopted network deployment, and the principal solutions introduced to provision context information.

Then, Chapter 6, Chapter 7, and Chapter 8 analyze the design and the implementation of real-world CDDIs. Chapter 6 presents our CDDI for emergency response scenarios, called *Reliable and Efficient COntext-aware data dissemination middleWare for Emergency Response (RECOWER)*, while Chapter 7 focuses on university smart campuses to present our CDDI *Scalable context-Aware middLeware for mobile EnvironmtS (SALES)*. Then, in Chapter 8, we focus on the realization of CDDIs for smart cities scenarios, by extending our distributed architecture with Cloud-based solutions for the sake of context processing. We remark that, although the presentation of our solutions is divided along these three chapters, solutions proposed in one scenario can be also adopted

in the other ones; this structure has been adopted for the sake of clarity, in order to present proposed solutions in the most appropriate scenario.

Finally, Chapter 9 and Chapter 10 end this dissertation by highlighting main thesis findings and by detailing still open challenges and future research directions, so as to better remark the main contributions of our work.

2. Context-aware Services in Future Mobile Systems

In the last few years, context-awareness, as the provisioning of the current execution context to the service level, has received an increasing attention, up to becoming a core feature of next-generation mobile networks. The capacity to gather and timely deliver to the service level any relevant information describing the provisioning environment, e.g., computing resources, current location, and user preferences, enables new compelling scenarios where services can self-adapt to ensure provisioning and improve user experience [1].

Much work has been done to introduce a unified definition of context awareness, capable of considering all the aspects useful to perform service adaptation [1, 2, 5, 6]. Unfortunately, such generalizations usually led to extremely broad definitions, difficult to apply and to manage in real-world scenarios, with the main outcome that, at the current stage, there is no widely accepted definition. At the same time, several research works pointed out the need of provisioning context to mobile devices with agreed quality levels, useful to ensure correct, timely, and meaningful adaptations. Also this area is characterized by contradictory and partial definitions, usually related with specific types of context-aware services [3, 4, 7].

Main goal of this chapter is to introduce all needed background material and to clearly state all main definitions used in the remainder of this dissertation, as well as addressed main deployment scenarios. Section 2.1 presents some examples that clarify the great potential of context-aware services in everyday life. Section 2.2 introduces the definition of context with its associated categorization, while Section 2.3 details the Quality of Context (QoC) notion. Section 2.4 presents the main peculiarities of next generation mobile networks, so to better justify the need of additional research in this area. Finally, Section 2.5 remarks the main motivations of this thesis work.

2.1. Context-aware Services

Context is a fundamental basic issue but also an intrinsic and hidden concept in our everyday life [1]. We continuously and implicitly process information coming from our own physical surroundings to automatically react and adjust our behavior. Even more, human beings are able to process complex and hidden context information, such as people mood and emotions, to adapt their own reactions, for instance, by reducing the pitch of the

voice, dynamically changing the distance with the interlocutor, and so forth.

Context-aware computing research strives at bringing this awareness to the computing world. For instance, when applied to mobile systems, context awareness allows novel scenarios where services can dynamically adapt according to time-varying and unpredictable conditions, usually consequence of user mobility. To clarify the great potential of context-aware scenarios, let us introduce few examples.

Every year, thousands of tourists visit the beautiful monuments in our city, Bologna; of course, tourists do not know the city center, and do not usually have easy ways to discover the most attractive locations to visit. Different tourists may rate attractions in different ways, depending on their historical background, and could be interested in different types of museum, monuments, etc. In this case, a *context-aware tourist guide* [8] would be extremely suitable to provide useful descriptions of the monuments, as well as ratings and comments left by previous tourists. Such a service can exploit both tourists and monuments profiles, as well as current localization information, to suggest downtown tours. In a similar way, it could exploit aforementioned context information to recommend close restaurants matching particular cuisine preferences.

In addition, Bologna features the oldest university in the occidental world. Thousands of students are currently enrolled in the different degree courses offered by our university; they usually spend several hours in the university area, close to the city center, where most of the university buildings are located. In this case, *smart campus services* could greatly enhance the social experience of our students, for instance, by recommending social events of interest, possible friendships with students sharing common interests, and so on [9, 10]. In addition, our university hosts several exchange students that want to carry on abroad studies. Those students usually need some time to settle in our city since they do not know the university area and do not have acquaintances and friends. By using current localization information, a smart campus service could automatically guide them in Bologna downtown, so to find university buildings in an easy way. Also, by exploiting student and place profiles, it could suggest possible friendships with both Italian and other exchange students, so to ease integration processes with local people, social customs, etc.

All above examples clarify the significance of context-aware services in mobile computing scenarios. Context can comprehend several and heterogeneous information, mainly used to characterize involved people and physical places in previous examples. It comes without saying it that additional context information, related to used computing devices and resources, are also extremely useful to tailor service provisioning and to avoid

user dissatisfaction, e.g., by avoiding the visualization of extremely complex and detailed web pages on a smartphone with a display of few inches. To settle the required terminology and clearly define the context information considered in mobile context-aware scenarios, next section delves into context definition and classification details.

2.2. Context Definition and Classification

Context is now a very wide meaning word, that may also express several and different senses according to specific scenarios and authors. From a merely practical viewpoint, *context identifies the aspects the designer considers useful to model the environment where a particular service is deployed and executed; such aspects are usually used at runtime to trigger appropriate service adaptations*. With a more theoretical connotation, several authors in the past introduced their own context definition.

To the best of our knowledge, the oldest and most referenced context definition for mobile computing scenarios has been presented in [1], where authors say that "*Three important aspects of context are: where you are, who you are with, and what resources are nearby. Context encompasses more than just the user's location, because other things of interest are also mobile and changing. Context includes lighting, noise level, network connectivity, communication costs, communication bandwidth, and even the social situation*". In [2], authors supply a broader definition, saying that "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*". In [11], authors write that "*elements for the description of this context information fall into five categories: individually, activity, location, time, and relations*". Finally, [6] defines context as a four-dimensional space composed by *computing context*, *physical context, time context,* and *user context*.

Although these definitions may seem different due to adopted viewpoints, they actually agree upon the main context aspects considered to trigger service adaptations. For the sake of clarity, in the remainder we adopt the context definition presented in [6] since capable of covering the main context aspects with a straightforward classification. In addition, it shares important similarities with the definition of [1], hence we can safely assume that it is overall well-accepted by the research community. Next subsections clarify each single context dimension, namely computing context, physical context, time context, and user context, by also introducing examples of useful runtime adaptations.

2.2.1. Computing Context

The computing context dimension deals with all those technical aspects related to computing capabilities and resources. It has a two-fold aim since it captures both local device aspects, such as network connectivity and bandwidth, display size and resolution, etc., and distributed ones, such as printers and servers in physical/logical proximity.

First, the computing context captures all those heterogeneities usually present in mobile environments. Traditional context aspects, such as display size and network bandwidth, are already widely used in commercial products. For instance, both Google and Facebook dynamically adapt to the current characteristics of mobile devices, Web clients, and connectivity; also, several video streaming services, such as Youtube, adapt video quality to device capabilities, mainly screen resolution, to save computational resources. For the sake of battery preservation, in [12], the authors introduce a new algorithm to reduce LCD backlight, while accounting for image distortion; the proposed schema considers the peculiarities of the specific LCD adopted by the device, as well as of displayed images, to reach a good tradeoff between image quality and battery savings. Focusing on context-aware Web services, in [13] authors present a new conceptual framework, made by a modeling language and automatic code generation tools, to ease the design, the implementation, and the deployment of context-triggered adaptation actions.

Second, computing context also takes into account the different resources that a mobile device encounters while roaming, such as printers, displays, connectivity opportunities, etc. [1]. In Always Best Connected (ABC) systems [14, 15], mobile devices melt together signal quality, battery status, pricing, and additional configuration policies, to select the best connectivity opportunity between available ones; connectivity selection is usually performed by using computing context aspects. Similarly, services that exploit close computing resources are attractive to enable impromptu collaborations between mobile users and devices, with no need of user-initiated reconfigurations. A smart printer service for university campuses can automatically find and redirect print commands to the closest printer, so to avoid students roaming in university buildings to find available printers [1].

2.2.2. Physical Context

The physical context dimension accounts for all those aspects that represent the real world, and that are usually accessible by means of sensors deployed in the surrounding environment. Absolute device and user locations are notable basic examples of physical

context; other interesting aspects include people speed, traffic condition, noise level, temperature, and lighting data. Physical models and laws, such as cinematic laws useful to predict future physical states of the mobile system, belong to this context dimension. We remark that, due to imprecise sensing techniques and stochastic nature of physical processes, context aspects belonging to such context dimension are usually very prone to errors; hence, particular attention has to be paid in triggering service adaptations, for instance, by applying low-pass filters and forecasting techniques to received context data, in order to prevent wrong adaptations [3].

For the sake of clarity, let us introduce few examples where the physical context dimension is considered. First and foremost, traditional navigation systems adopted in modern cars use current location information, provided by the Global Positioning System (GPS), to compute route directions to the final destination; they surely represent the most common example of context-aware service based on physical context. Staying with vehicular scenarios, [16] proposes a context-aware solution for intelligent traffic lights that, based on current traffic jamming conditions, adapts red/yellow/green times to improve road condition. Similarly, many solutions use attributes belonging to this context dimension to perform environmental monitoring, such as monitoring systems that use video sensors deployed on vehicles to detect plate numbers of suspicious cars and prevent collisions with wild animals [17]. Finally, considering more local adaptations, in [18] authors present an adaptive method to scale LCD backlight according to the lighting conditions of the surrounding environments, so to preserve device battery.

2.2.3. Time Context

The time context mainly deals with the time dimension, such as time of a day, week, month, and season of the year, of any action performed into the system. Actions can be referred either to real-world, e.g., human beings, or computing actors, e.g., software agents. We remark that these context aspects can be of two main types, namely sporadic and periodic. Sporadic events are used to model unexpected actions triggered occasionally, even only once. Instead, periodic events model actions that present themselves in a repeated and predictable way. Of course, more complex time context events, for instance, based on event sequence, number of events in a particular time period, and so forth, are possible and should be properly supported by the system [2]. We also recall that the implementation of particular primitives is not straightforward in distributed systems as time synchronization between different devices is usually not enforced.

To consider real service scenarios, a sporadic event can be associated to a temporary network congestion that automatically triggers an adaptation action meant to reduce the quality of the video streaming provisioned on the mobile device. A periodic event can be consequence of an activity detection system that, for instance, automatically switches off the cell phone ringing tone every day, from 11pm to 7am, to avoid waking up users. Finally, complex time events can take into account, for instance, repeated network congestion occurrences; in this case, by monitoring congestion frequency and inter-arrival times, a context-aware service can decide to switch between different connectivity opportunities for the sake of user satisfaction.

2.2.4. User Context

The user context dimension contains high-level context aspects related to the social dimension of users, such as user's profiles describing main interests and cuisine preferences, people located nearby, current social situation, and so forth [19]. Above all, let us remark that we are considering distributed mobile systems where users and devices interact among themselves, hence, as also noticed by [20], each node context contains 1) an individual dimension, descending from its own egocentric view (e.g., user profile and preferences); and 2) a social dimension, descending from the awareness of being part of a whole system (e.g., people in physical proximity and current social situation).

Several systems already adopt this kind of context aspects to perform automatic recommendation and situation-based adaptation. For instance, in [21], authors exploit co-localization patterns to infer common interests between users and to recommend new friendships; social events are described through profiles, later used by the system to understand if two events, although different, share commonalities in addressed topics. Similarly, [22] presents a situation-aware service that, by aggregating context information coming from running services, physical sensors deployed in the environment, and people profiles, understands if a work meeting is taking place; in that case, the service can automatically switch off the cell phone ringing tone until the end of the work meeting.

2.3. Quality of Context (QoC)

Aforementioned context aspects are widely used in literature to adapt services at runtime, so to fit the current execution environment characteristics and to make them satisfactory for final users. Notwithstanding the huge potential of these scenarios, important challenges have to be carefully addressed during the real-world realization of

context-aware services. Among others, the quality of the context data is fundamental since erroneous data can misguide service adaptations; at the end, such a reduced usability can mine the widespread adoption of this class of services since users will be upset by wrong and unexpected reconfiguration actions.

Above all, it must be noted that some intrinsic errors, associated with sensing techniques, cannot be avoided; some context aspects, such as lighting condition and temperature of a room, are acquired by physical sensors that introduce approximations due to limited sensor resolution. Similarly, context data produced by logical sensors, such as user profiles fetched by a database, can present partial and incomplete information, as well as errors when produced by reasoning techniques; hence, even if usually more polished than physical ones, these context data also need proper management mechanisms to assess their quality.

Consequently, the notion of *Quality of Context (QoC) – defined as the set of parameters expressing quality requirements and properties on context data (e.g., precision, freshness, trustworthiness, …)* – is fundamental to control and manage all the possible context inaccuracies [3, 4]. Several research works analyzed the usage of QoC-based mechanisms useful, for instance, to solve conflicts between context data produced by different sensors [23]. Although it is widely recognized that QoC-based management mechanisms are essential in real-world context-aware scenarios, current research works present conflicting definitions and objectives, usually tailored for specific services and types of context data. Hence, the remainder of this section aims to clearly state the notion of QoC adopted in this dissertation, as well as the main quality attributes considered in both context data and distribution process.

2.3.1. Definition and Motivations

Similarly to context awareness in itself, QoC is an extremely blurred concept with several meanings. Some works have already studied both context quality parameters and their effects on context data distribution; following a temporal order, we now briefly detail the principal works in this area, starting from the oldest one.

In [3], the authors define QoC as "*any information that describes the quality of information that is used as context information. Thus, QoC refers to information and not to the process nor the hardware component that possibly provide the information*". This definition decomposes the quality problem along three main directions, namely 1) *the quality of the physical sensors*; 2) *the quality of the context data*; and 3) *the quality of the*

delivery process; according to authors' definition, QoC deals with only the second dimension. More specifically, quality of context data contains the following main attributes: 1) *precision*, to describe how exactly the provided context information represents the reality, so to account for sensor resolution; 2) *probability of correctness*, to denote the probability that a context information is correct, and not wrong due to internal sensor problems; 3) *trustworthiness* to rank if the context data producer can be trusted; 4) *resolution*, to denote the granularity of the information in terms of spatial/temporal constraints; and 5) *up-to-dateness*, to account for context information aging. Although trustworthiness could seem overlapped with probability of correctness, it accounts for a different aspect: trustworthiness is used by a context producer to rate the quality of the other entity that originally produced the context information, while probability of correctness is attached by the context producer itself according to its own local beliefs.

Moving toward more recent definitions, in [7], authors introduce a QoC notion based on four main parameters: 1) *up-to-dateness*, to deal with context aging; 2) *trustworthiness*, to indicate the belief we have in context correctness; 3) *completeness*, to account that context data could be partial; and 4) *significance*, to express the absolute priority of context data. Apart from some similarities with the classification proposed in [3], here authors introduce two new attributes, namely completeness and significance, and present algorithms to solve context data conflicts at runtime; a conflict resolution policy exploits a weighted combination of quality parameters, in order to select the data to be saved [23, 24]. In addition, the significance parameter introduces connotations related to the context data delivery, thus being in contrast with the definition of [3].

Furthermore, [25] exploits a standard ISO vocabulary for measurements to define a new QoC framework. Authors critically analyze the ISO standard to understand what attributes can be adopted in real-world context-aware computing systems. First of all, the authors argue against the usage of the accuracy attribute as a means to rate how close a measurement is to the real value. In fact, the automatic evaluation of the accuracy by a context producer is unfeasible since the context producer should know the real value of the context information itself; of course, since there is no practical way to have such information, accuracy cannot be ever automatically calculated by the system itself. Instead, as suggested by the adopted ISO vocabulary, authors consider precision to rank how the results from measurement sensors are repeatable; for instance, although a temperature sensor can be inaccurate by reporting higher temperature values, if such values always present the same error in respect of real values, the precision of the sensor is

high. Hence, context data generated from a producer with a good precision exhibit relative dynamics that closely follow real context values. In addition, authors consider *up-to-dateness*, *resolution*, and *trustworthiness* with meanings similar to the homonym attributes proposed by [3], and detail a new algorithm to dynamically evaluate trustworthiness parameter based on users' feedbacks.

Finally, in [26], authors focus on the quality of information produced in wireless sensor networks. They consider the notion of operational context to facilitate the dynamic binding between applications and sensors; in particular, they exploit the 5WH principle (why, when, where, what, who, and how) to describe both application requirements and produced data, and use these attributes to handle context data distribution at runtime. In addition, they consider *spatial* and *temporal relevancy* of produced data, in order to establish if a piece of information has temporal and spatial properties compatible with application requirements, and propose two metrics to evaluate such quality parameters.

In conclusion, a widely-accepted QoC definition is still missing. Different authors focused on specific aspects and presented their own QoC framework. At the same time, some research works tend to separate the quality attributes associated with the context data from the ones associated with the context data delivery process [3]; differently from them, we think it is not always possible to clearly separate these dimensions, hence, we are more prone toward extended QoC definitions and contracts useful to clearly manage the whole context delivery process from producers to consumers.

However, at least a common thought can be highlighted in all the aforementioned works: QoC is not about requiring perfect context data, such as context data with the highest possible up-to-dateness, but about having and maintaining a correct estimation of the data quality. In fact, context data without a proper quality characterization are both dangerous and useless, since service reconfigurations could be completely misled by low quality data. QoC is also motivated by the following main scenarios [3]:

- *QoC agreements* - As Quality of Service (QoS) permits service consumers and producers to negotiate their requirements at acceptable service levels by considering the network available underneath [27], when several entities cooperate to the provisioning of context-aware services we need proper quality contracts to tailor the interactions between context data producers and consumers. For instance, in many scenarios, the same context data can be produced by different sources, each one leading to particular QoC attributes; in this case, QoC is essential to perform an accurate selection of the final context producer to use.

Furthermore, whenever a context consumer receives new context data, it can take more accurate decisions, spanning from using the data, weighting it if partially incomplete, or completely discharging it due to the extreme low quality. QoC agreements can be also dynamically negotiated at runtime depending on available resources; for instance, due to network bandwidth limitation, a context producer and a context consumer can be forced to use a low data sending rate, thus experiencing limited context up-to-dateness.

- *Adaptation of context data reasoning* - A fundamental task of every context-aware system regards the production of new context data out of low-level information coming from single sensors. In this case, context data quality is fundamental, and must be considered during reasoning to assess if generated context data will be meaningful at all. In addition, it will be necessary to also determine the quality of the produced context data. Apart from context data reasoning, also simple filtering techniques, usually meant to filter low quality data for the sake of scalability and to interpolate historical information to highlight trends, have to consider data quality information to avoid wrong droppings and deductions.
- *Adaptation of context data distribution* - Every context-aware system needs to memorize and distribute context data to interested mobile nodes. If context producers and consumers agree on particular QoC objectives, it is possible to introduce novel adaptive data distribution solutions to increase system scalability by reducing the runtime management overhead. For instance, if context consumers can accept context data with low up-to-dateness, we can supply cached context data to save resources; of course, this adversely affects QoC but, if correctly managed, does not reduce the quality stipulated with users.
- *Reconstruction of the context-aware service behaviour* - As context data automatically trigger services adaptation, the usage of low quality data has a significant impact on the context-aware users' experience. If only low quality data are available, it is important to warn the user to let him better evaluate both the current situation and the possible dangers introduced by automatic context-aware reconfigurations and suggestions. In addition, if the context-aware service presents wrong behaviours, it is important to easily understand the real cause, e.g., either a traditional programming bug or low quality context data.

- *Privacy policies* - As several context-aware services make use of sensitive context data, QoC is important to restrict the access to personal and sensitive information. Access policies can depend on the quality of the context data; for instance, while a user will not consider the distribution of coarse-grained localization information as a privacy violation, he will surely want to restrict the distribution of localization information with spatial resolution in terms of hundreds of meters. Hence, QoC attributes and constraints are useful to detail and enforce appropriate privacy policies on context data.

To conclude, QoC-based context data distribution is a core requirement to enable appropriate runtime system management in real-world scenarios. Also supported by aforementioned related works and requirements, we remark that there surely exist two main quality directions to consider: *one related with the quality of the data, the other related with the quality of the context data distribution*. Hence, in the remainder, we adopt a broader QoC notion that, apart from data quality attributes, considers the quality of the context data distribution (e.g., data delivery time, reliability, …) to ensure the availability of the context data with the right quality, in the right place, and at the right time [3].

2.3.2. Quality of Data

Each context data instance has to be associated to and described by proper quality attributes, mainly to evaluate the usefulness of such piece of context information. We carefully consider the previous works on data QoC [3, 4, 7, 25], but we extend them to better fit our view. Now, we point out the data QoC framework adopted in the remainder of this thesis.

In finer details, our QoC-based context data framework considers the following quality attributes. First of all, *context data validity* rates the compliance of the context data with the field of validity of the specific type; for instance, a time context data must conform to the Gregorian calendar format. Second, *context data precision* evaluates the degree of adherence between real and sensed values of a context data. We agree with some previous works in the area, such as [25], that it is not possible to dynamically evaluate the precision of each data instance as this supposes to know the real value of the context data; hence, our notion of precision exploits information coming from the resolution offered by sensing devices. For instance, focusing on precision, ultra-wide-band-based localization data are more precise than standard GPS-based ones. Third, *context data up-to-dateness* takes care of data aging, so to express how the usefulness of particular data changes over

time. Differently from previous approaches, this is not a simple timestamp but rather a more complex law that can also introduce non-linear functions between the absolute time elapsed from data generation and the current up-to-dateness value. For instance, the up-to-dateness quality attribute associated with location information of a fixed resource, e.g., GPRS antenna, is not greatly affected by time elapsing, while the same attribute for a mobile entity, e.g., a mobile device carried by a user, can quickly degrade according to current mobility patterns. Fourth, *context data significance* ranks the relative importance of a context data. This attribute is mainly used by the management infrastructure during conflict resolution phase and storage replacement to keep the most important data. Finally, similarly to the work of [3], *context data trustworthiness* is used to rate the quality of the other entity that had originally produced the context information, and *context data probability of correctness* is attached by the context producer itself at generation time to rate the local beliefs he has in sensor correctness.

All the above QoC parameters should be taken into account in the QoC agreement specified at the service level and, at the same time, used by the context data distribution to measure and achieve the fulfillment of the QoC requirements.

2.3.3. Quality of Delivery Process

QoC has to consider the quality of the distribution process to ensure user satisfaction. Context data have to be dispatched from producers to consumers according to negotiated quality levels. For instance, a stale data is both useless, since it will be probably discharged at the consumer node, and dangerous, since it could lead to wrong adaptations if used by the consumer. In addition, since context data distribution usually takes place through best-effort wireless infrastructures that could introduce delays and droppings, thus leading to additional inaccuracies in the context received by mobile devices, it is also necessary to negotiate proper priority and reliability levels.

Hence, in our opinion, QoC has to deal also with dynamic aspects related to the delivery of the context data to single consumers. To support this model, context data have to be tagged with proper quality parameters that, differently from the ones treated in the previous section, depend on the consumer requesting them. Let us clarify the meaning of the last sentence with two examples. A context data instance has an up-to-dateness value that mainly depends on the time elapsed from data generation; it does not matter which context consumer requires it, as long as the instance respects the QoC constraints imposed by the consumer. Hence, the up-to-dateness quality attribute will change only as

consequence of time elapsing, and not according to the consumer requesting the data. Instead, let us consider two different consumers that require context data from the same producer with different priority levels. For instance, the same context data describing the people contained in a particular physical place can be required by both friends finder service and by police for monitoring purpose; of course, the context data sent in answer to police requests must be dispatched with higher priority. Hence, these quality attributes on the context data distribution process depend on the consumer requiring them, and are dynamically determined at runtime according to the context data request.

Even if Buchholz et al. already highlighted the importance of quality attributes on the distribution process [3], they clearly divided QoC, related to context data only, from QoS, connected instead to the distribution process. As stated before, we do not agree with this sharp separation as some quality attributes are actually entangled, thus presenting correlations between their values. For instance, some data quality parameters are intrinsically dynamic, as their value depends on the time elapsed from data generation. In this case, since delivery delays affect these quality parameters, context data matching consumer QoC requirements at the producer side could not match them when actually received by the consumer. Hence, the assumption that data QoC parameters do not depend on the quality parameters of the distribution process is violated [3]. Also, Buchholz et al. introduce a contradiction since they say that "*QoC refers to information and not to the process nor the hardware component that possibly provide the information*"; but the QoC trustworthiness parameter clearly depends on the process (e.g., the intermediate entities involved in context data routing) that provides the information to the final consumer.

In conclusion, in the remainder we consider some main QoC parameters related to the dispatching process, namely 1) *data retrieval time*, to capture the maximum time limit between the request of context data and the actual delivery to the consumer; and 2) *priority,* to consider differentiated priorities depending on the consumer requesting the data, so to reconfigure intermediate dispatching processes.

2.4. Next Generation Mobile Networks

Although context-aware solutions have been adopted in different research areas, context-awareness reaches its maximum usefulness in mobile systems. In fact, context awareness permits mobile services to dynamically and efficiently adapt both to the current situation, such as current physical place and social activity, and to the challenging and highly variable deployment conditions typical of such mobile environments (device

resource scarcity, unreliable and intermittent wireless connectivity, …). The significance of context-aware capabilities in mobile computing is also evidenced by the plethora of research efforts proposed in the last years in this area [28-37].

Before proceeding further, it is necessary to clearly define our main deployment scenario. In fact, the realization of real-world context-aware services in mobile systems requires a deep understanding of many technological details and includes several non-trivial operations, spanning different layers and also depending on executing platforms. For instance, different wireless technologies can limit and constraint the possibility of interactions between mobile nodes, while different operating systems for mobile devices can introduce peculiar limitations on the usage of connectivity opportunities. In addition, context delivery can introduce a high management overhead, especially when the system scales up to thousands of devices. In this case, the availability of specific technologies, such as ad-hoc communications between mobile devices in physical proximity, can greatly help the implementation and real-world realization of context distribution mechanisms. Also, there exist novel routing protocols that opportunistically exploit node encounters to spread data into the mobile system with a delay tolerant fashion [38]. Although such routing mechanisms are not common yet as further studies are required to better assess their runtime performance, we believe that they will soon become fundamental to perform message routing and data distribution in mobile systems for the sake of wireless infrastructure offloading. Finally, in the near future, mobile systems will integrate with Cloud architectures [39], with the main goal of dynamically offloading heavy computations, such as video and image processing, so to save device battery.

For the sake of clarity, Figure 2.1 reports an example of future mobile system to highlight the main associated peculiarities, namely heterogeneous devices, heterogeneous wireless technologies and modes, etc. Starting with lower-level technological details, and moving toward higher-level routing stacks and Cloud solutions, next subsections detail what we mean with and expect from future mobile systems.

2.4.1. Heterogeneous Environments

The overwhelming success of mobile devices, wireless communications, and novel mobile development environments is paving the way to the anytime from everywhere connectivity view of pervasive computing. Notwithstanding several advantages of the new scenario, the design and the implementation of a context delivery support are very complex tasks because of both the high deployment scenario heterogeneity (in terms of

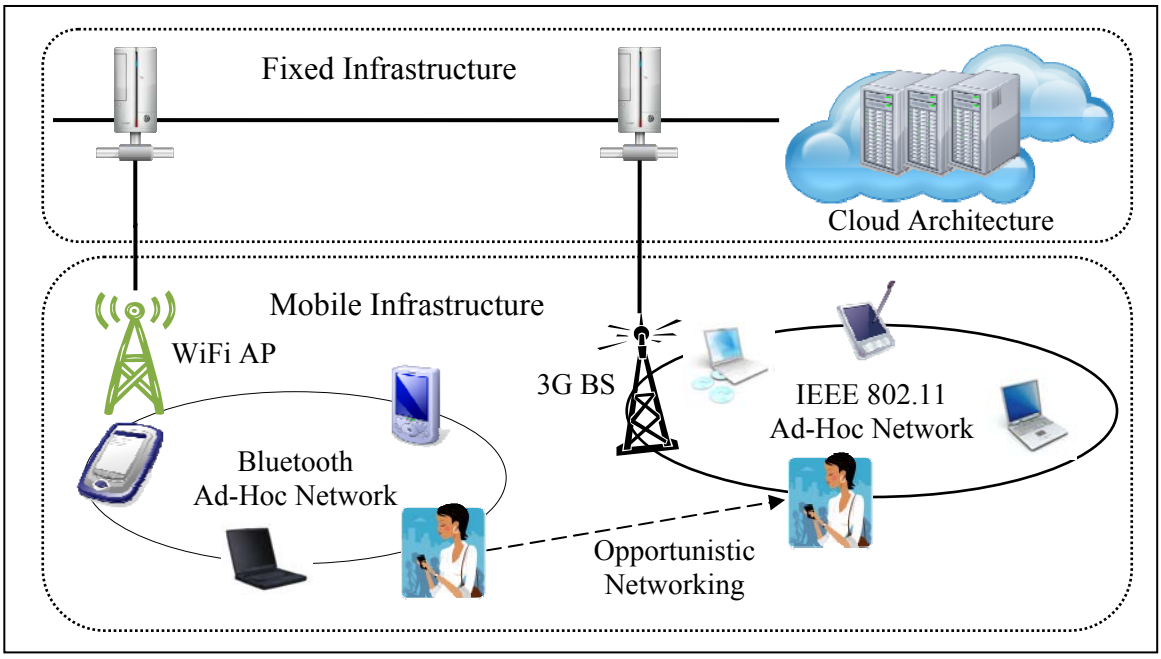

Figure 2.1. Future Mobile Systems.

client characteristics, employed wireless technologies, etc.) and its inherent resource scarcity.

First, future mobile systems will feature extremely heterogeneous devices, spanning from full-fledged laptops to resource-constrained and battery-powered PDAs. A big challenge is that different mobile development platforms, such as Java 2 Micro Edition, Windows Mobile, Apple iOS, and Android, implement different features and suggest different logical architectures for service development. There are already a few good papers analyzing the main differences of these platforms, and we refer interested readers to those works for a thorough analysis [40]. Here, we want to underline that all the main mobile platforms offer poor support of some basic features out of programmer direct control, such as garbage collection and object serialization, and typically introduce high dependencies on the mobile operating system. That latter problem affects especially cross mobile development platforms that, notwithstanding their goal of being independent of the hardware and software available underneath, often suffer from inconsistent implementations, especially for low-level and system-dependent libraries, such as wireless networking APIs.

Second, since most of the devices that participate to context distribution are portable, battery-powered, and resource-constrained, particular attention should be paid in monitoring and controlling final overhead, to keep it low with respect to available resources. Let us note that context delivery is a continuous background process that introduces long-running management overhead in CPU utilization, memory usage, and

network bandwidth. From a general viewpoint, interesting context data should be continuously delivered to mobile devices, so to detect context-based situations and trigger proper reconfigurations; of course, partial offloading of context processing to fixed infrastructure is feasible as well, but possible only when there are reliable and not expensive means to exchange data between mobile and fixed servers. That latter assumption is not always ensured if we consider mobile devices roaming in a city with 3G network connectivity only.

To conclude, a context management infrastructure for future mobile systems has to consider both heterogeneity and resource scarcity to avoid the introduction of unfeasible management overheads that, in their turn, could lead to the slowdown of context-aware services wide acceptance [6].

2.4.2. Hybrid Infrastructure-based/Ad-hoc Communications

While mobile users keep asking for novel classes of services that require transferring huge amounts of data over fixed wireless infrastructures, such as video streaming services, current wireless communication technologies are suffering this increased bandwidth pressure, and are struggling to get behind the over increasing traffic demands of next years. It is important to note that, in December 2009, mobile data traffic surpassed voice one on a global basis, and is expected to double annually for the next five years [41] [42]. As main consequence, several European and US operators are considering the end of unlimited data plans to cool this surging demand [43].

Hence, there is a remarkable attention toward hybrid distributed architectures, capable of jointly exploiting both infrastructure-based and local ad-hoc communications between mobile devices, to reduce infrastructure traffic and increase system scalability. On the one hand, mobile devices, usually called relays, can distribute data to neighbours in physical proximity without additional load on the wireless fixed infrastructure; from the wireless infrastructure point-of-view, only one initial transmission, useful to transmit the data from the fixed infrastructure to the relay, is required to deliver the data to the entire one-hop neighbourhood of the relay. On the other hand, since the coverage of wireless fixed infrastructures is not always ensured, relays enacting as bridges can extend infrastructure coverage through multi-hop ad-hoc communications. In addition, ad-hoc communications enable data exchange between close devices everywhere; in specific deployment scenarios, such as the ones resulting from natural disasters or terrorist attacks, this type of communication is essential since wireless infrastructures could have been also damaged.

From the technological point-of-view, different wireless standards already allow ad-hoc communications between mobile devices. The most remarkable ones are surely IEEE 802.11a/b/g (WiFi) and Bluetooth (BT), currently available on a large portion of commercial mobile phones and PDAs. Although with different transmission ranges and bandwidth, WiFi and BT interfaces can be adopted to realize hybrid wireless networks. At the same time, mesh networks, namely hybrid wireless networks where intermediate nodes act as relays for the sake of infrastructure coverage and throughput, belong to an extremely active research area that, apart from featuring important standardization efforts (such as the IEEE 802.11s amendment), is proposing optimized MAC and routing protocols that can also suggest interesting solutions for hybrid mobile networks.

Unfortunately, hybrid infrastructure/ad-hoc architectures introduce additional complexities that have to be properly addressed before the real-world production phase. Nodes acting as relays are usually resource-constrained mobile devices that greatly suffer from the additional management duties. Hence, proper incentive mechanisms need to be designed and realized to avoid final users acting selfishly. At the same time, if mobile devices are directly involved into the routing, we need proper resource reservation and security mechanisms to ensure service provisioning. Both throughput and fairness over multi-hop routing paths are hard to ensure, while message confidentiality and integrity are fundamental in real-world service provisioning. All these additional requirements further complicate the software stack needed to support hybrid networks in real-world scenarios.

2.4.3. Opportunistic and Intermittent Connectivity

Real-world service provisioning scenarios usually assume connected topologies where both the sender and the destination are available and willing to communicate at the same time. This time synchronization greatly limits possible interactions between nodes, and is usually not ensured in more dynamic mobile networks, such as Mobile Ad-hoc NETworks (MANETs) and Vehicular Ad-hoc NETworks (VANETs) [44, 45], where nodes continuously join and leave the system. In addition, apart from 3G cellular networks that usually ensure always-on city-wide coverage, connectivity with medium-range wireless networks, such as a wireless network composed by WiFi hotspots deployed in a university campus, is usually intermittent during user roaming. As main consequence, opportunistic networks, intended as self-organizing networks able to exploit all the communication opportunities available in the physical environment, are slowly emerging as one of the core mechanisms required by future mobile systems.

To be more specific, let us introduce a short example based on the context-aware tourist guide for the Bologna downtown. For the sake of service provisioning, a first seminal solution is to use WiFi hotspots to periodically offload important context data to mobile nodes, but this solution limits data availability since context data can be distributed only when mobile nodes are reachable via those fixed WiFi hotspots. However, mobile nodes can opportunistically exploit neighbours to carry context data requests into the system. While roaming, some mobile nodes could get closer to WiFi hotspots, where they can offload the requests on behalf of peers and store resulting data. With a similar mechanism, context data can be routed back and delivered to requesting nodes, thus enabling service provisioning also when there is no fixed infrastructure available. Unfortunately, such scenarios do not allow the introduction of strict quality guarantees since node mobility is generally difficult to predict.

Following this main research direction, several academic research works have started to consider the opportunistic usage of mobile devices to perform data distribution into the system [38, 46]. Acting as data carriers, mobile nodes can distribute data to close devices by ad-hoc communications, with no additional overhead on wireless fixed infrastructures. Also, mobility is a fundamental means to speed-up data distribution into the system, by exploiting random and intermittent interactions between devices. Hence, although several challenges still need to be addressed, opportunistic networks will soon become an important part of traditional provisioning scenarios in mobile systems.

2.4.4. Integration with Cloud Architectures

Cloud technologies enable the dynamic provisioning of computational resources, with a pay-per-use billing model [39, 47]. At the current stage, different big players, such as Amazon, Google, and IBM, are adopting such technologies to offer computational resources to third parties, with the main goal of reducing data center operational costs. Above all, as Cloud solutions allow the rapid and dynamic scaling of provisioned resources, they well fit all those scenarios characterized by high fluctuations of demands. In fact, we do not need to overprovision IT infrastructures according to the worst-case load scenario but, if required, we only have to ask for additional computational resources to a Cloud.

Mobile devices suffer severe resource limitations that make it unfeasible to execute heavy computational tasks, e.g., video stream processing, aboard. Similarly, context data have to be stored and processed to enable service adaptation; that can introduce a high

management load, hence, it may be useful to dynamically offload such computations to a Cloud reachable through wireless fixed infrastructures. We also remark that the capability of dynamic resource provisioning well matches mobile systems, where users randomly join and leave, thus perhaps leading to high load variations along the day. At the same time, it must be kept in mind that wireless fixed infrastructures may suffer bandwidth limitations, thus reducing the possibility of data transfer between the mobile infrastructure and the Cloud. Moreover, data transfer from/to wireless fixed infrastructures may lead to faster battery depletion. All these issues must be carefully considered to enable a fruitful integration between context-aware mobile systems and Cloud solutions.

2.5. Motivations of the Thesis

In the past years, much work has been done to enable context awareness and to ease the diffusion of context-aware services. One of the most cited and important papers on context-aware services in mobile systems by Schilit et al. is [1]; from that year, several works discussed the main mechanisms useful to enable context-aware facilities in mobile systems, as well as local support mechanisms to provision context to the service level. At the same time, several middleware solutions have been designed to transparently implement context management and provisioning in the mobile system.

However, at the current research stage, we feel that an in-depth analysis of the context data distribution, namely the function in charge of distributing context data to interested entities, is still missing. Previous works mainly focused on the support mechanisms required to provision context to services from a more local viewpoint, e.g., context representation, reasoning, and local delivery, without considering the enormous complexities arising from the deployment of such services in large-scale mobile systems. One of the main goals of this thesis is to envision, design, implement, and test novel distribution primitives and mechanisms capable of ensuring context provisioning in a scalable and reliable way.

In addition, as clarified in previous sections, future mobile scenarios will feature extremely heterogeneous devices, capable of interacting through different wireless modes (infrastructure-based/ad-hoc), and with the possibility of opportunistically exploiting both devices and wireless connectivity opportunities available in the physical proximity. Previous works analyzed context data distribution in small scale environments, such as houses and university buildings, where always-on WiFi WLANs can continuously provide context information to mobile devices with no particular issues. Due to the adopted

assumptions, we think they widely miss the need of quality-based context data distribution schema able to trade off quality constraints with the introduced management overhead. Hence, the impact of all the peculiarities introduced by future mobile systems on context distribution primitives is not clear, and deserves additional research to assess main potentialities and shortcomings.

To conclude, starting from the core assumption that only effective and efficient context data distribution can pave the way to the deployment of truly context-aware services, in this thesis we aim to put together current research efforts to derive an original and holistic view of the existing research on context-aware systems. We present a unified architectural model and a new taxonomy for context data distribution, by considering and comparing different distribution primitives in deployment scenarios that jointly exploit heterogeneous wireless standards and modes. To better assess the technical soundness of our analysis, we then consider three main deployment scenarios, and we study the main consequences of network architectures on context-aware service provisioning. Finally, we conclude the thesis by drawing and identifying important directions of future work.

3. Context Data Distribution in Mobile Scenarios

Context-aware capabilities ground on the continuous delivery of important context data to interested mobile nodes. From a general viewpoint, context data distribution is an extremely complex function that has to deal with different management phases involved in context provisioning to the service layer, namely context data representation, storage, aggregation, distribution, notifications to running services, etc. At the same time, the main peculiarities of future mobile systems also call for proper management solutions to efficiently deal with resource constraints, heterogeneous wireless standards and devices, etc.

To transparently tackle all these issues and to ease the diffusion of context-aware services, we need proper middleware solutions, called *Context Data Distribution Infrastructure (CDDI)* in the remainder, capable of hiding all the main phases involved in context management [48]. In other words, context-aware services should only have to produce and publish context data and to declare their interests in receiving them from the CDDI, while the CDDI takes over distribution responsibility and transparently executes specific management operations to distribute context data.

In this chapter, we delve into the details of context data distribution infrastructures for mobile systems. Section 3.1 clarifies the main issues such CDDIs have to address. Section 3.2 presents our main design guidelines for scalable context provisioning in mobile systems, while Section 3.3 introduces an in-depth discussion of the context data life cycle. Then, Section 3.4 discusses related work on context-aware systems, so to better point out the current state-of-the-art. Finally, Section 3.5 ends this chapter by drawing intermediate conclusions.

3.1. Main Issues

Real-world CDDIs for large-scale mobile systems have to transparently address several issues related to the delivery of huge amounts of context data to resource-constrained mobile devices. From a general viewpoint, we categorize the main involved issues along three main directions, namely *heterogeneous and resource-constrained devices*, *context data management and delivery scalability*, and *quality-based context provisioning***.**

Future mobile systems feature resource-constrained mobile devices that require

continuous access to their context while roaming. These devices are mainly battery-powered, and with strict CPU/memory constraints that will not allow the execution of complex CDDIs. Similarly, most of the context reasoning algorithms will introduce prohibitive costs, thus requiring the offloading of such computation to fixed servers. Apart from device limitations, wireless networking introduces additional constraints that have to be carefully considered during CDDI design. Next generation mobile devices will feature numerous wireless interfaces, thus enabling different connectivity opportunities and data transfer facilities with differentiated tradeoffs between bandwidth, energy consumption, etc. While 3G/4G cellular network interfaces are extremely suitable for voice service provisioning due to the ensured coverage, WiFi and BT interfaces enable ad-hoc communications between mobile devices in physical proximity. However, even if available, all these interfaces cannot be always powered on due to excessive battery consumption. In conclusion, CDDIs for future mobile systems have to consider such resource constraints: they must be resource-aware in respect of device local resources and communication opportunities to properly handle context provisioning.

Moreover, when we consider large-scale networks made by thousands of devices, for instance city-wide services, both the memorization and the delivery of context data introduce important CDDI scalability bottlenecks. Such systems feature thousands and thousands of sensors that continuously pump new context data. Context data production rates strictly depend on the described context aspects: while temperature and pressure sensors can produce new data with a period in the order of seconds, we expect that logical sensors associated with user profiles will produce new data with a period in the order of days. Although sensors can be directly deployed on mobile devices, if the limited resources do not allow the processing of the raw context data on the devices themselves, the usefulness of having these data suddenly drops since they must be offloaded to fixed infrastructures. At the end, the storage of all these data introduces a high overhead, that becomes even worse if the CDDI has to offer access to historical data. Similarly, the distribution of such context data from/to the mobile nodes introduces not negligible bandwidth overhead. The usage of 3G/4G networks for the sake of context data distribution would be probably prohibitive due to the limited bandwidth; it must be considered that the primary goal of cellular networks is to offer voice services, hence, data traffic over such infrastructures should be kept as small as possible to avoid excessive interferences with important core services.

Last but not least, as introduced in Section 2.3, QoC attributes, both on the context

data and on the distribution process, play an important role to prevent useless and noisy adaptations. Real sensors come with sensing errors that can lead to wrong adaptations; even worse, the CDDI can distribute context data referring to the same aspect with contradictory values, thus introducing an ambiguity that must be solved through quality-based approaches. At the same time, the usage of unreliable wireless infrastructures can lead to additional errors in the context provisioned to mobile nodes; both packet droppings and long transmission delays caused by temporary network congestions can lead to the usage of wrong and/or stale context information. All these issues become even more difficult to address if we consider distributed architectures that jointly use infrastructure-based and ad-hoc communications. In this case, the enforcement of particular quality constraints, such as maximum data retrieval time, can involve complex resource reservation and negotiation protocols, spanning multi-hop routing paths based on intermediate mobile devices; at the same time, such routing paths can also include hops based on heterogeneous technologies, thus making the prediction of path stability and transferring times more difficult [49]. Hence, quality-based context provisioning in large-scale mobile systems, perhaps based on heterogeneous wireless standards and modes, is not easy to enforce from the CDDI viewpoint; instead, it introduces a great deal of complexity since the impact of these quality constraints on the distributed context distribution function could lead to not trivial reconfigurations.

3.2. Design Guidelines

While a lot of research has been done as regards the design, the realization, and the deployment of context-aware middleware solutions, most of the previous works focused on rather small-scale deployments, with the main goal of studying the local middleware infrastructure useful to support context provisioning to service level. Instead, in the very last years, an increasing number of systems is requiring context provisioning in large-scale wireless systems; to mention few examples, VANETs and smart cities feature different context-aware services, such as accident prevention and environmental monitoring, that require to distribute huge amounts of context data with city-wide scope [17, 50]. As discussed in Section 3.1, here the context data distribution becomes a fundamental concern, requiring innovative solutions to efficiently deal with all the peculiarities of such large-scale scenarios.

Hence, to effectively support such context-aware services, starting with low-level issues connected to mobile deployment scenarios, and moving to more high-level and

complex ones associated with context data management, we claim the significance of six fundamental design guidelines: 1) *adaptation to mobile and heterogeneous environments*; 2) *efficient context data life cycle management;* 3) *context data production/consumption decoupling*; 4) *context data visibility scopes enforcement*; 5) *cooperative context data delivery*; and 6) *QoC-based context data distribution*.

The CDDI should support mobile heterogeneous wireless scenarios. Mobile nodes executing context-aware services move in and out, even randomly, thus introducing variations in context needs; hence, the context data distribution has to promptly adapt to mobility, in order to distribute only currently needed context data. At the same time, the CDDI should deal with heterogeneous systems, including nodes with different computational capabilities, wireless interfaces belonging to different standards, and different wireless modalities. While the usage of heterogeneous wireless standards enables multiple transmissions happen at the same time with limited interferences, the usage of heterogeneous wireless modes lets the CDDI trade off context availability and management overhead: fixed wireless infrastructures offer reliable context access but introduce tight limitations on available resources, while ad-hoc communications let close peers exchange data without additional overhead on deployed wireless infrastructures. In conclusion, to exploit all the possibilities offered by future mobile systems, the CDDI has to adapt to currently available resources to increase system scalability, while preventing saturation conditions.

The CDDI should efficiently manage the whole context data life cycle, starting from data generation to removal [51]. In particular, it has to implement efficient context data aggregation and filtering techniques, by also taking care of final context data removal when necessary. Aggregation techniques are useful to reason about raw/fine-grained context data, so to obtain more high-level and concise information. Instead, filtering techniques enable to shape context data distribution, so to reduce the management overhead according to service needs. Both these techniques should be supported in a distributed manner so, for instance, to filter the distribution of a data as close as possible to the node that had generated it. Moreover, the CDDI should offer some guarantees of availability by influencing the degree of replication of the data into the system. Hence, the CDDI has to automatically manage context data during the whole life cycle, by offering efficient aggregation and filtering techniques, and by memorizing context data at multiple places for the sake of reliability.

The CDDI should transparently route produced context data to all the interested

consumers connected to the mobile system. To increase system scalability and context data availability, context data production and consumption should be possible at different times (time decoupling), and producers and consumers do not have to know each other (space decoupling); hence, communication should be asynchronous and anonymous among producers and consumers, similarly to what already happens for traditional pub/sub systems [52]. In fact, both space and time decoupling favour the asynchronous execution of context-aware entities, that can inject and retrieve context data according to their own needs.

The CDDI should introduce and enforce differentiated visibility scopes for context data. Context data usually have a limited visibility scope that depends on physical/logical locality principles. For instance, physical context of a place is likely to be required only by nodes in the same place (physical locality); similarly, context data associated with a particular event are likely to be required only by its participants (logical locality). Hence, context data intrinsically have visibility scopes that the context data distribution should enforce to avoid useless management overhead. To effectively avoid context data storage and distribution bottlenecks, the CDDI should adopt decentralized and hierarchical storage architectures that exploit both physical and logical locality principles on context data to make them available as close as possible to interested consumers.

The CDDI should realize cooperative context delivery mechanisms to increase both context availability and system scalability. Mobile nodes should cooperate among themselves to store and distribute the context data associated with the physical place where they are currently in; by doing so, newly arrived mobile nodes can retrieve interesting context data from neighbours through ad-hoc links, without any requests forwarded to the fixed infrastructure. Cooperative context distribution is also useful to refine context data; for instance, several devices, equipped with temperature sensors, can exchange readings to merge them by means of aggregation operators, such as average, median, etc., so to have a better assessment of the context data quality [53]. In addition, the CDDI should also introduce opportunistic network facilities to let mobile devices route context requests on behalf of others.

The CDDI should introduce and enforce QoC constraints to enable correct system management. QoC constraints on context data, used to specify the quality of received context information, are useful to setup proper filtering operators [3]. In addition, as real-world wireless systems present frequent topology changes, limited delivery guarantees, and intermittent disconnections, QoC constraints on context data distribution allow

enforcing data delivery with particular timeliness and reliability guarantees. Finally, since the CDDI can be deployed in distributed architectures where several servers, each one with its own local context data repository, process and route context data, we have to consider that context data could be available in multiple and conflicting copies into the system [54]. Therefore, as context data consistency is costly to handle in large-scale systems, it is advisable to avoid strong consistency by preferring best-effort approaches driven by QoC constraints.

From aforementioned design guidelines, it comes without saying it that context data management is an extremely complex task, requiring several mechanisms from both a local and a distributed viewpoint. Hence, for the sake of clarity, next section introduces an in-depth presentation of context data life cycle, by highlighting and discussing the main involved phases; in this way, we aim to better justify the main mechanisms a CDDI for mobile systems has to introduce.

3.3. Context Data Life Cycle

One of the main management duties of a CDDI is to handle the whole context data life cycle, from context data production to removal. From a general viewpoint, context data life cycle is made by different phases, that can be also executed repeatedly to refine context information, and sometimes with no strict temporal order.

After the initial sensing of the raw data, new context data are introduced into the system. These new data can be delivered to context-aware services, and can be stored by the CDDI to ensure persistency and later access; they can be filtered according to QoC constraints, as well as other filtering operators based on context data value, to reduce the number of stored and distributed data; they can be aggregated with pre-existing context data to produce high-level context information, such as merging together temperature and pressure data to understand if the weather is rainy; finally, they can be distributed to interested mobile devices to enable service adaptations.

Although some life cycle phases present intrinsic strict temporal orders (e.g., context data production must be the first one), other intermediate phases can mix them together. For instance, the aggregation/filtering steps and their temporal order depend both on the needs of mobile services and on the availability of resources; the CDDI can store either raw context data or their aggregated/filtered counterparts, hence, storage phase has no strict temporal ordering with other phases; and so forth. At the same time, context data can go through some phases multiple times: a *cyclic phase* needs to be triggered in different

time instants since its usefulness and produced results change during time. For instance, context data aggregation is performed in a time-triggered manner to infer new context data based on the knowledge accumulated so far; similarly, context data delivery has to be performed multiple times if the set of interested mobile nodes changes.

With aforementioned observations in mind, we decided to adopt a context data life cycle model based on five main phases, namely *context data production, context data storage, context data aggregation, context data filtering,* and *context data delivery*. Figure 3.1 shows a general overview of the main phases involved in context data life cycle; the following subsections better detail each one of them.

3.3.1. Context Data Production

Context data production is the initial phase through which all the context data have to pass. This phase comprehends both the real access to the sensors in charge of producing the raw data, and an elaboration phase aimed to represent the context data according to specific representation techniques [55]. Before the real injection into the system, additional elaborations can be performed at the producer side, for instance, to apply low-pass filters, evaluate quality attributes, etc.

Following the definitions introduced in [48], *sensors* are usually categorized in three main categories: *physical, virtual,* and *logical*. Physical sensors include the many hardware sensors available today to capture physical data, such as temperature, pressure, humidity, lighting condition, etc. Virtual sensors acquire raw context data from software services; for instance, a virtual sensor can publish the current user situation by looking at his calendar, keyboard and mouse activities, as well as running services on his laptop. Similarly, a virtual sensor can fetch user profiles available on the Internet, such as the ones

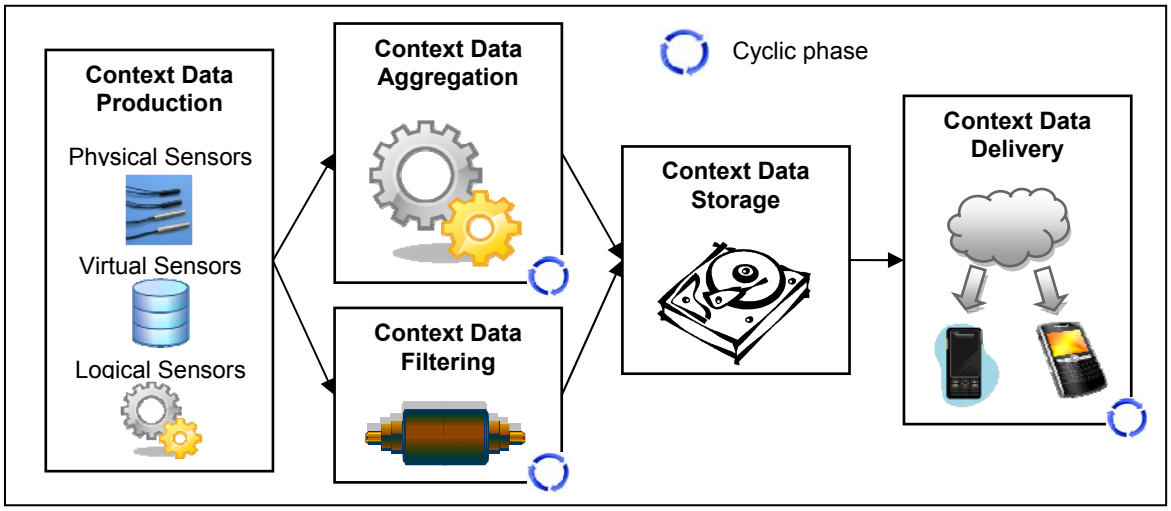

Figure 3.1. Context Data Life Cycle Overview.

adopted by traditional social networking services, to better describe user interests. Finally, logical sensors combine context data coming from other sensors to produce high-level and more polished context data. For instance, user localization detected by means of GPS sensors is not always reliable since the user could have left the mobile phone in his car, hence, for the sake of precision, it is advisable to make use and merge together information coming from different sensors, such as an GPS readings, video reporting the user in specific places, Web browsing and login activities. Of course, logical sensors include aggregation operators and capabilities to produce new context data.

After sensing, context data are represented by means of a proper representation technique, such as key-value pairs, XML-based documents, ontology-based solutions, etc. [55]. The different approaches, with associated pros and cons, will be better analyzed in Section 4.2.1. For now, we note that it is a producer duty to describe injected context data with additional management attributes, such as the ones used for lifetime and quality management. In addition, as sensors could be deployed on resource-constrained mobile devices that connect with fixed infrastructures through bandwidth-limited connections, the producer itself can apply filtering operators to raw context data so, for instance, to slow down data injection into the system. Finally, if suitable due to battery constraints, data prediction techniques can be used to forecast future values and schedule raw data samplings only when remarkable changes are expected.

3.3.2. Context Data Storage

In the context data storage phase, context data are stored into the distributed CDDI architecture to ensure context data availability and persistency. This phase can be omitted as not all the context data have to be stored; for instance, some context data should be provisioned only to the local device due to privacy reason, hence, they are not distributed at all. When required, the context data storage phase takes care of storing data into the system, by triggering all the required management mechanisms and coordination protocols. Storing context data is not as simple as it may appear since the system scale requires proper additional management mechanisms, such as context data caching and replication, to ensure system scalability. For the sake of clarity, let us briefly clarify that data copy into the system can happen by means of either caching or replication techniques [56, 57]: caching techniques reactively maintain data in response to requests, and usually keep them until deleted by local replacement operations, mainly due to memory saturation [58, 59]; instead, replication techniques proactively copy local data to remote nodes, and

keep them until explicitly deleted [60, 61].

As mentioned in Section 3.2, a CDDI should be able to exploit cooperative context data distribution mechanisms. In view of this guideline, context data can be either cached or replicated into the system. If each mobile node has a small context data repository shared with close neighbours by ad-hoc links, the CDDI can effectively reduce the management traffic toward the fixed infrastructure, so to foster scalability. However, both caching and replication techniques introduce additional management issues to be considered and properly managed. Especially if applied to mobile systems, those techniques have to adopt explicit mechanisms to handle the context data stored on mobile nodes in physical proximity, in order to avoid that they memorize the same set of data; in fact, if data repositories in the same physical area share a large set of common data, the CDDI will store a reduced number of different data, thus potentially leading to a higher number of requests forwarded to the fixed infrastructure.

Apart from the adoption of caching/replication techniques, a fundamental point is that the CDDI has to organize the data storage to bring context data closer to the interested nodes, so to reduce data retrieval times and runtime overhead. Toward this direction, both physical and logical locality principles offer good hints to organize the context data storage. A straightforward application of the physical locality principle is to store context data produced by sensors deployed on a particular physical environment only on the nodes currently within that physical environment; for instance, temperature and pressure readings associated with a room are probably considered interesting only by the nodes in the room, or close to it, and such interest usually decreases with the distance from the production point. In other words, the higher the distance from the production point, the lower the interest usually expressed by the mobile nodes; hence, the CDDI should adopt hierarchical storage architectures to match such principle, so to keep the data as close as possible to their production point. Similarly, the logical locality principle suggests to tailor context data storage depending on the interests expressed by the nodes. For instance, consider a physical place, e.g., a university lab, that is usually crowded of Computer Science students; although some context data, such as data associated with a meeting of the Network Research Group currently taking place on the other side of the building, are produced far away, they should be stored and made available into the lab to simplify the distribution to incoming students. Hence, the usage of the logical locality principle should also affect the storage architecture adopted by the CDDI.

Finally, service requirements can also greatly affect the complexity of the CDDI

storage architecture. Let us consider a simple context-aware service that performs high-level reasoning on the people usually co-located in a particular physical place. In this case, the service needs historical data about user presence, as well as profiles and additional information on the people carrying those devices. Similarly, if the service wants to foresee the number of attendees of a particular event, it has to reason on what happened in the past editions of such event. Hence, both these examples require context data history facilities, and this introduces additional complexities and management overhead to the CDDI storage architecture.

3.3.3. Context Data Aggregation

Physical sensors usually provide raw data associated with physical phenomena, such as temperature, pressure, acceleration, localization, etc. Virtual sensors enable access to more high-level context data, such as place and user profiles, usually fetched by database. Finally, logical sensors aggregate context data, coming from different and heterogeneous sensors, to produce new context data out of raw sensor readings.

Context data aggregation is a fundamental mechanism in real-world CDDI. Due to the huge amount of possible context directions, it is practically impossible to statically and manually define all the interesting context aspects that can be considered in a large-scale system. At the same time, many context data cannot be statically defined as they are consequence of particular runtime situations, such as people in physical proximity; such context data claim for continuous updates that must be carried on automatically by the system. Hence, context data aggregation phases are fundamental to capture additional knowledge about the system; Artificial Intelligence (AI) provides techniques, as well as standard logic-based representations and inference engines, that can simplify the usage of such techniques.

Differently from filtering operators, whose main goal is to tailor context data distribution for the sake of scalability, aggregation operators do not selectively drop input context data, but instead produce and inject completely new data as output. Hence, by using a metaphor, filtering operators are mainly adapters useful to connect two pipes with different sizes (e.g., with different flow rates), while aggregation operators are put in parallel and combine flows from different pipes to generate a completely new flow.

Since aggregation operators need access to multiple context data to properly work, one of the main CDDI issues is to ensure the availability of all the required data in the context data storage used by the aggregation operator. In addition, proper CDDI

mechanisms are required to evaluate both management and QoC attributes of new generated data. For instance, a very simple example regards the evaluation of the data lifetime attribute that, by adopting a pessimistic approach, can be set to the lowest value carried by initial context data instances; more complex approaches are instead required to evaluate QoC parameters, such as up-to-dateness, where the usage of simple merging functions (minimum/maximum, average, etc.) is more difficult to justify.

Finally, let us remark that context aggregation operators usually require access to the history of input context data to better guide the aggregation process. In fact, historical values can be used to predict future values and highlight trends in input values. All this additional information is useful to avoid the injection of new context data that, by considering only the latest, perhaps erroneous, context data input, can present significant errors.

3.3.4. Context Data Filtering

Sensor sampling, and subsequent context data production, can happen extremely frequently, with periods in terms of seconds. At the same time, the provisioning of all these data to mobile devices is not always feasible since it can saturate available resources, both in terms of CPU and memory, and in terms of wireless bandwidth. The usefulness itself of delivering similar or slightly different context data is questionable too; context-aware services usually trigger adaptation actions in response to sensible changes (for instance, in terms of localization and people in the current room) rather than in response to small context changes, perhaps not perceived by final end users. Hence, the CDDI has to introduce proper filtering operators, useful to tailor context data production according to service needs, while striking to reach a balance between context completeness and runtime overhead due to context data processing and transmission at the mobile node.

In finer details, context data filtering operators are fundamental in real-world CDDIs for large-scale systems, but they must be carefully handled to reach and enforce negotiated QoC. In fact, such operators produce partial and imprecise context views at the context-aware service that, in its turn, by reasoning according to the received context data can trigger incorrect adaptations. QoC contracts have a fundamental role in this direction as they enable the correct configuration of the involved filtering phases. Not all the context-aware services will weigh the access to particular context data in the same way; for instance, a smart printer service for university campus is interested only in sensible location changes, such as entering and leaving a room, while a context-aware guide

requires extremely accurate localization information, as well as spatial orientation of the user; similarly, accident prevention services for VANETs require a very precise characterization of the current physical neighbourhood to detect potential dangerous situations, while buddy finder services can surely tolerate coarse-grained localization data and perhaps erroneous friend suggestions with no important consequences.

Finally, similarly to what usually happen with video streaming services, the CDDI has to monitor the runtime behaviour of the system to assess whether negotiated QoC levels can be enforced. An initial negotiation phase lets producers and consumers agree about QoC objectives on context data, in order to setup filtering operators; then, at runtime, if the CDDI cannot ensure required QoC levels, for instance, due to the saturation of a link or intermediate node overloads, it has to notify running context-aware services in order to make them aware of the reduced QoC.

3.3.5. Context Data Delivery

The context data delivery phase takes care of automatically delivering injected context data to all those entities that have expressed any form of interest in them. Different forms of interaction between the CDDI and the mobile nodes, e.g., either push- or pull-based, are available, with different tradeoffs between management overhead and perceived QoC.

During this phase, it is overwhelming important to consider both context data interests expressed by the mobile node and QoC objectives. The CDDI has to automatically drop context data that will not respect QoC objectives at the destination node, so to prevent useless overhead; out-of-QoC context data droppings should also happen as close as possible to the producer node, in order to prevent the triggering of intermediate filtering and aggregation operators that, in their turn, will lead to useless management overhead.

Let us remark that solutions adopted to address this phase has a great impact on previous ones, that ground upon it to inject new data into the system (context data production), and to retrieve context data to aggregate and filter according to context-aware service needs (context data aggregation and filtering phases). Without delving into deeper details, which will be clarified in Chapter 4, the usage of distributed solutions to address this phase, an inescapable choice in large-scale settings, automatically calls for distributed ones also for previous phases: for instance, if context-aware services define complex context filtering operators, the CDDI can distribute single operators to different servers for the sake of load balancing; similarly, complex aggregation operators can exploit intermediate, already aggregated, context data to reduce final CDDI overhead.

Finally, this phase presents some significant differences from previous ones. First, solutions adopted here must be distributed: while production, aggregation, and filtering phases refer to more local computations, the distribution phase in large-scale systems must be distributed between different nodes, also spanning different and heterogeneous networks, such as fixed nodes in different local area networks, mobile devices reachable by wireless infrastructures, etc. Second, being closely related with underlying network deployments and topologies, it is widely affected by them: adopted coordination protocols, as well as data distribution mechanisms to perform the context delivery (e.g., pub/sub architectures, multicast primitives, etc.) [52, 62], cannot be easily ported to different deployment architectures. Finally, from the CDDI viewpoint, this is the very last phase where the CDDI has control over the context data; after it, context data have been transmitted to mobile devices, that autonomously process and use them according to their will.

3.4. Context-Aware Systems Related Work

After an in-depth presentation of the context data life cycle, in this section we present a selection of the most important CDDIs for mobile systems. For each solution, we supply a short introduction and we clarify the main peculiarities introduced by authors. For the sake of readability, the following presentation has no pretence of being exhaustive; interested readers can refer to the few survey works, including ours, existing in literature [5, 6, 48, 63, 64] for an in-depth analysis of existing context-aware systems. At the same time, we note that additional references will be provided in the following chapters, when we will analyze single case studies. Now, we present related works in increasing time order, from the oldest to the newest ones, to better remark the evolution of this research area by presenting the main directions still under investigation as the last ones.

Starting from oldest works, they mainly focused on innovative frameworks and software mechanisms to provision context information to running services. Context Toolkit is one of the most significant works on context awareness [36]. It mainly considers the deployment of context-aware services from a local viewpoint, by introducing proper software mechanisms to locally handle context information. Context Toolkit is based on the concept of widget, i.e., a reusable component in charge of context data production and consumption. Apart from widgets that directly access sensors, it is possible to define meta-widgets, namely widgets that aggregate different context data to produce higher level context information. Similarly, MobiPADS focuses on context provisioning and

notification to running services [28]. It introduces the concept of Mobilet, namely an entity that provides a service and that can be migrated at runtime between different environments. Each Mobilet is implemented as a traditional client/server application, and can be dynamically migrated to allow both computation offloading and code migration. Mobilets can be chained to implement more complex services. One of the most important peculiarities of this work is that services are associated with proper profiles that detail context-based reconfigurations; MobiPADS takes care of the provisioning of context data associated with the local mobile device, such as battery status, to enable Mobilets runtime adaptation. Finally, CARISMA also focuses on context-aware service adaptation and associated mechanisms [37]. Similarly to MobiPADS, in CARISMA context-aware services supply proper profiles that can be modified at runtime by means of reflection mechanisms. Since each service provides a local profile, perhaps by detailing actions to trigger as consequence of specific context situations, conflicts can arise: for instance, if the device is going out of battery, one service can require turning off the wireless interface, while another one tries to keep it on for the sake of service provisioning. To solve such problem, CARISMA adopts a micro-economic approach, where each service rates each possible alternative profile; the profile that maximizes the satisfaction of all the local services is finally selected.

From aforementioned related works, we conclude that initial research efforts mainly focused on the usage of context information at runtime, as well as effective and efficient software mechanisms to locally handle and provision context data. They focused on rather small scale deployments, where both context data availability and distribution do not present particular issues. Differently, more recent works started recognizing the main issues associated with context data distribution when 1) it is not possible to assume direct interactions between mobile devices and sensor nodes; and 2) the system grows by including several mobile devices running multiple context-aware services.

EgoSpaces focuses on the usage of tuple spaces to perform context data distribution between close mobile devices connected through ad-hoc links [31]. It exploits the notion of mobile agent, i.e., a mobile entity that contains a private tuple store and that can migrate. Each agent can operate over multiple views that include context data coming from agents in the physical proximity of the device. To limit the scope, each view is defined through metadata constraints defined on both data and resources. Although this work seems more related to software engineering in general, it also considers context data delivery mechanisms: it highlights the importance of asynchronous interaction between

context data producers and consumers, and uses tuple spaces to support such model. Pervaho, instead, distributes context data to mobile nodes by using a publish/subscribe-oriented interface [32]. The authors exploit a Location-based Publish/Subscribe System (LPSS) to impose localization-based constraints: each publication and each subscription has a visibility scope, and a publication is delivered to an active subscription only if publisher and subscriber lie in the intersection of these two scopes. In this way, Pervaho implements location-based filtering, thus enforcing the physical locality principle in context delivery. From a rather implementation viewpoint, the LPSS is realized by means of a fixed server, reachable through a wireless fixed infrastructure, that executes a centralized JMS publish/subscribe server [65].

Moving to systems designed for larger scenarios, SOLAR exploits a peer-to-peer fixed infrastructure built by different physical servers, called Planets, to deliver context data to roaming mobile nodes [33]. This solution exploits the Context Fusion Network (CFN) that provides data processing facilities, both aggregation and filtering operators. Complex context data processing tasks are expressed through operator graphs, defined in terms of producers, consumers, and real operators. Context services can finely tailor received context data by supplying proper policies to the CFN: for instance, filtering techniques based on context data content are natively supported by the platform. In addition, authors introduce an adaptive mechanism that, by monitoring the queue of the context data to be delivered to a mobile device, automatically adapts context delivery rates to prevent overload conditions. Hence, we can safely argue that SOLAR exploits QoC requirements detailed by the service level to adapt context data delivery at runtime. By always considering large-scale settings, HiCon is a conceptual framework useful to manage large amounts of context data in extremely decentralized scenarios [30]. It exploits both physical and logical locality principles to reduce the amount of context data transmitted into the distributed system. In addition, HiCon adopts a three-level tree-like architecture where each node performs partial context data aggregation and filtering before transmitting context data to peers and/or to the level above. In brief, in HiCon authors focus on the implementation of complex context processing operations in a distributed manner, in order to reduce the runtime traffic and increase system scalability.

While these last works clearly point out the increased research efforts on context delivery infrastructures for large mobile settings, we conclude this section by presenting two very recent works that highlight an increasing attention toward delivery infrastructures able to dynamically self-adapt. COSINE is a software framework aimed to provision

context data in completely decentralized ad-hoc networks [66]. It exploits an XML-based context data model, and consumers use XPath queries to subscribe to a particular producer; when different producers are available, COSINE ranks them according to QoC parameters, and sends the subscription to the best one with no need of service intervention. Also, the adopted approach has an interesting outcome when a context subscription needs data from multiple sources: if there exists an aggregator service that already collects all the required data, the subscription is directly routed to it; otherwise, the initial subscription is automatically split in a set of fine-grained subscriptions, one for each required context producer. Finally, MobEyes addresses data harvesting in urban monitoring scenarios by exploiting vehicular networks [17]. Context data are initially produced by sensors, either deployed on vehicles or on fixed infrastructures. Then, vehicles store collected data and distribute them into the system to ensure availability: every time two vehicles come into contact, they exchange data by flooding, meaning that each vehicle downloads all the unknown data from the other one. Data retrieval is based on mobile software agents that carry consumer requests and travel the network to harvest as much interesting data as possible. Also, MobEyes exploits bio-inspired algorithms able to mark already harvested regions, in order to ensure efficient and fast data harvesting by driving agents toward information-productive regions.

To conclude, we can assess that local context data management issues, such as representation and notification to running services, have been already widely addressed in the past. At the current research stage, there is an increasing attention toward efficient mechanisms for context data delivery in large-scale settings. In addition, when the system scale is very large, such as a city, adaptive delivery mechanisms, mainly driven by QoC constraints, are extremely important as they permit to trade off context quality and runtime management overhead. Hence, also due to the novelty of these efforts, additional research is required toward comprehensive CDDIs for large-scale scenarios that, by transparently managing runtime resources and QoC constraints, are able to optimize the delivery process, so to support efficient context provisioning while granting system scalability.

3.5. Chapter Conclusions

In this chapter, we discussed the main issues introduced by the design of a CDDI for large-scale mobile settings. To ensure scalable and quality-based delivery, we decided to adopt few important design guidelines; some of them are strictly related with low level data transmission, e.g., joint usage of heterogeneous wireless standards and modes, while

others focus on the management of the context data into the distributed architecture, e.g., context data production/consumption decoupling and locality principles. We remark that one of our guidelines, namely QoC-based context data distribution, is also the means to enable runtime resource management and data distribution adaptation, with the main goal of increasing system scalability.

Then, we considered the whole life cycle of the context data. The CDDI has to offer a complex software stack to effectively handle context production, storage, aggregation, filtering, and distribution. Such software stack is even more complex when we deal with large-scale mobile settings, where mobile devices can randomly join and leave the system, thus potentially triggering continuous reconfigurations of the CDDI.

Finally, we introduced the state-of-the-art on modern context-aware systems. We presented several works, starting from oldest to the newest ones, to better highlight current research directions in this area. At the end of our analysis, we concluded that, although CDDIs for small deployments have been largely investigated in the past, additional work is required to fully adopt context-aware capabilities in large-scale systems. At the same time, to the best of our knowledge, an in-depth analysis of the possible design choices available at the CDDI level is still missing. In the next chapter, we will further analyze modern CDDIs and we will propose a new CDDI logical model useful to better understand all the main involved components, as well as associated responsibilities.

4. Context Data Distribution Infrastructures: Logical Model and Design Choices

CDDIs for large-scale wireless systems have to deal with different and heterogeneous management duties, spanning from data storage to delivery to mobile nodes, under strict resource constraints and unpredictable mobility. At the same time, the adopted distributed architecture widely affects the context data distribution, by introducing peculiar aspects, e.g., intermittent connectivity and limited context access, that further complicate the design of such solutions.

While previous chapters focused on background knowledge and related work, this chapter starts discussing the original contributions of this thesis. From our analysis, we derived a new logical CDDI model, with associated main layers and design choices. For each possible solution, we will discuss the main tradeoffs between feasibility and quality-based context provisioning. Finally, after a brief presentation of possible network deployments, we will discuss principal similarities and differences with pre-existing data distribution approaches in literature; that will strengthen our guidelines and design choices by highlighting the peculiar needs of context data distribution in mobile systems.

The chapter is organized as follows. Section 4.1 presents our CDDI logical architecture. Then, Section 4.2, Section 4.3, and Section 4.4 better analyze CDDI main layers, with the main goal of highlighting fundamental requirements, possible solutions, and inter-dependencies with adopted network deployment. In Section 4.5, we present main deployment solutions and we discuss what we believe are the main commonalities and differences between CDDI and pre-existing data distribution approaches. Finally, Section 4.6 ends this chapter with intermediate conclusions.

4.1. Context Data Distribution Infrastructure Main Layers

Both the heterogeneity and the complexity of the design guidelines presented in Section 3.2 claim for complex context data distribution solutions that transparently distribute context data to all the interested entities, while monitoring currently available resources and ensuring QoC constraints [3]. Since the wider the system scale, the higher the overhead introduced by context distribution, novel decentralized solutions are required to implement the context distribution function in large-scale wireless systems.

Above all, context data distribution systems are data-centric architectures that

encompass three main actors: context data producers (sources), context data consumers (sinks), and context data distribution function (see Figure 4.1). Context producers access back-end sensors and inject new context data into the system. Context consumers express their own context needs by using either context data queries (pull-based interaction) or subscriptions (push-based interaction); context data matching is the satisfaction of consumer requests, both query and subscription, to achieve a correct fulfillment of both types. Finally, the context data distribution function distributes context data by mediating the interactions between context data producers and consumers; for instance, it automatically notifies subscribed consumers upon context data matching. In the remainder, we use the expressions *"context producers"* and *"context sources"*, and *"context consumers"* and *"context sinks"*, interchangeably.

With a closer view to the organization, the only main phases executed directly by the service level are expressing context data needs and producing context data, that involves both sensor access and subsequent context data injection into the system. Then, the context data distribution function takes care of the other main phases, namely storage, aggregation, filtering, and delivery. Given the central role of the distribution function, its own efficiency is fundamental to ensure scalability. Directly stemming from our main guidelines, we adopt the internal architecture detailed in Figure 4.1: it contains two horizontal layers – *Context Data Management* and *Context Data Delivery*, starting from the uppermost to the lowest one – and one vertical cross-layer – *Runtime Adaptation Support* – better clarified in the following sections.

4.2. Context Data Management Layer

The Context Data Management Layer takes care of local context data handling, by defining context data representation, by storing context data, and by expressing processing

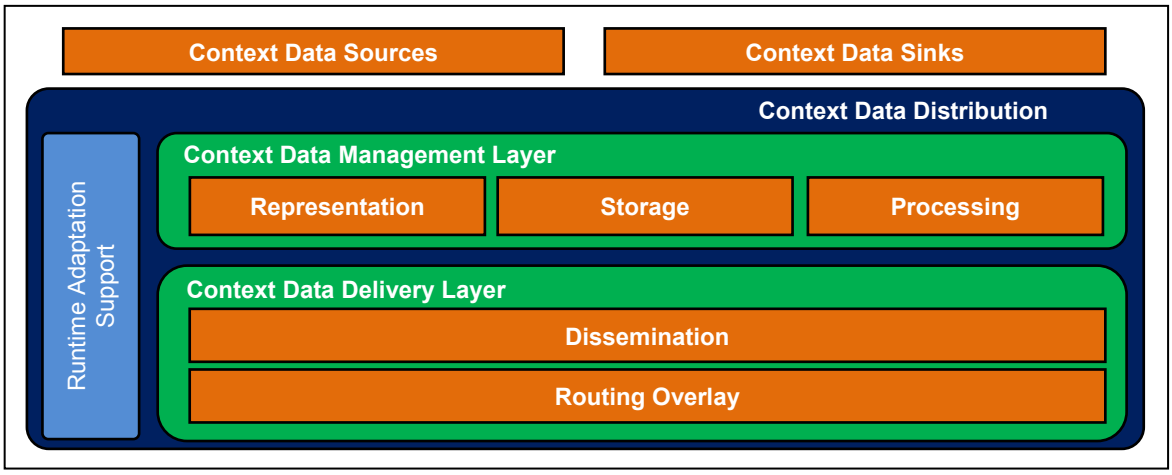

Figure 4.1. Context Data Distribution System Logical Architecture.

needs and operations. Context data representation includes all different techniques, spanning from simple and flat name-value pairs to ontology, proposed to represent context data at the CDDI level [55, 67]. Context data storage includes all the techniques to both cache and replicate context data into the distributed CDDI architecture, by also taking care of past context history. Context data processing includes 1) complex aggregation techniques (such as simple data matching, first-order logic aggregation, semantic-based techniques, …) to produce new knowledge from pre-existing context data; 2) simple filtering techniques to adapt context data distribution to currently available resources and QoC requirements, so to foster system scalability [48]; and 3) all the primitives useful to ensure context data security during the distribution process. Let us note that local context-aware services interact directly with this layer through their own sinks, which take proper management decisions according to expressed context needs. Local context needs are usually expressed by means of context data filters that also include data QoC constraints. QoC constraints, for instance based on data up-to-dateness, are 1) locally used to filter the context data supplied to the final services; and 2) remotely used to avoid the distribution of out-of-QoC data that will not be used by requesting node.

In the remainder, we discuss the main facilities of the context data management layer, by also detailing the main possible approaches with associated pros and cons. For the sake of clarity, Figure 4.2 briefly highlights the different solutions that can be adopted at each facility of this layer.

4.2.1. Context Data Representation

Several models have been proposed to represent context information; they differ in expressiveness, processing overhead, and memorization cost. Focusing on expressiveness, we divide context data models in *general* and *domain-specific*. General models, concerned with the generic problem of knowledge representation, offer a wide design space to enable the representation of any known service domain. Domain-specific models, instead, represent only data belonging to a specific vertical domain, and do not enable the specification of generic data; thanks to the reduced scope, these models usually specify complex data manipulation operations. Hybrid solutions, based on the usage of two or even more models, either general or domain-specific, are also feasible, but may require additional mapping functions to convert data from different models.

General models offer different degrees of formalism and expressiveness. Since model expressiveness relates to offered data operations, more complex models tend to supply

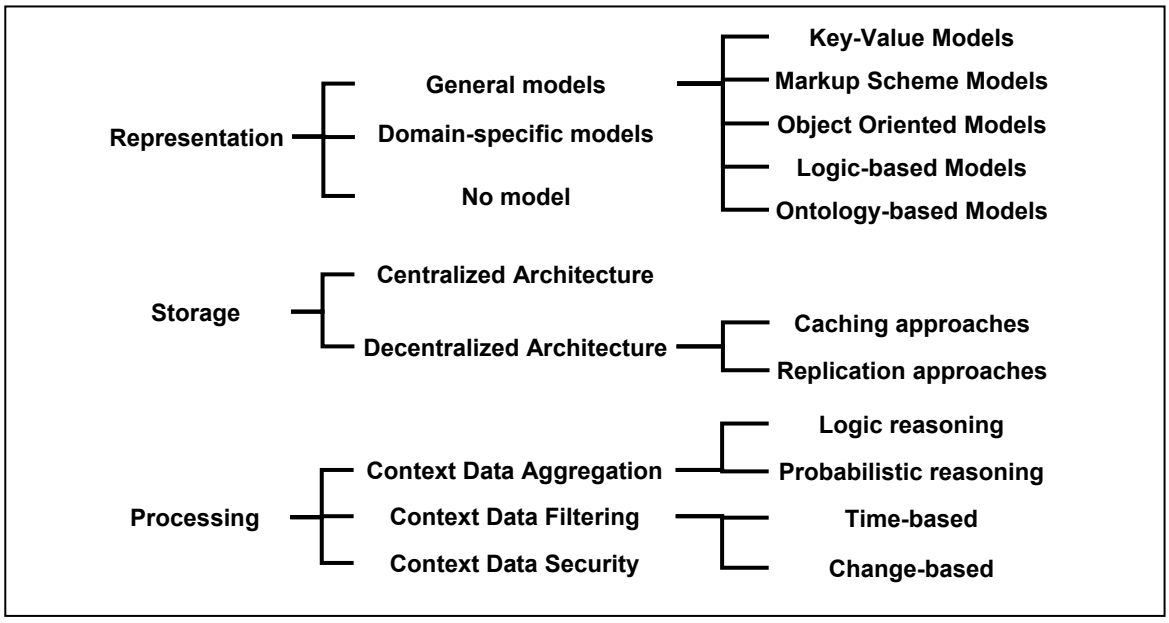

Figure 4.2. Taxonomy for the Classification of the Context Data Management Layer.

additional data operations, like aggregation operators to derive new context data and quality operators to specify and manage QoC constraints. With an increasing order of complexity, context data representation can adopt one of the following models [55, 68]: *key-value models, markup scheme models, object oriented models, logic-based models,* and *ontology-based models*.

Key-value models represent the simplest structure for modeling context by exploiting pairs of two items: a key (attribute name) and its value. Simplicity is the main reason for this approach popularity. Unfortunately, these approaches lack capabilities for structuring context data, and do not provide mechanisms to check data validity.

Markup scheme models use XML-based representations to model a hierarchical data structure consisting of markup tags, attributes, and contents. These approaches solve some of the limitations of key-value models; for instance, they support data validation by means of XML-schemas, and structured data definition via nested XML.

Object oriented models take advantage of the benefits of object-oriented approaches, typically encapsulation and reusability: each class defines a new context data type with associated access functions. Type-checking and data validity can be ensured during both compilation and runtime execution, while QoC elements can be easily mapped as other objects. In addition, these models ease interactions between services and context data, since the usage of the same abstractions provided by object-oriented programming languages simplifies the deployment of context handling code.

Logic-based models exploit the high expressiveness of logic formalism: context

contains facts, expressions, and rules, and new knowledge can be derived by inference. Traditionally, these models focus on inference mechanisms and provide also proper formalisms to specify inference rules. Unfortunately, they do not usually offer simple functionalities to deal with data validity; validation can be ensured, but associated rules are not straightforward and depend on the adopted type of logic.

Ontology-based models use ontologies to represent context, and take advantage of the capability of expressing even complex relationships. Data validity is usually expressed by imposing ontology constraints. By focusing on relationships between entities, ontologies are very suitable for mapping everyday knowledge within a data structure easily usable and manageable. In addition, the wide adoption of ontologies enables the reuse of previous works through the creation of common and shared domain vocabularies. Although ontology approaches seem very competitive, mobile environments usually avoid them since required computing resources, in terms of CPU and memory, are not acceptable for resource-constrained mobile devices.

As stated before, general models offer a great degree of freedom to represent everyday knowledge. Differently, domain-specific models are less flexible and, by focusing on a particular domain, introduce particular constraints on the data and on the relationships between them. On the bright side, this restricted flexibility enables the definition of more complex automatic aggregation operators. For instance, spatial data models are widely adopted by localization systems to represent both real-world objects location and relationships among them [69], such as containment and intersection; in the literature, there are also some standardization efforts that clearly define data and spatial query format, and such level of standardization has greatly simplified the definition and the implementation of automatic data management tools. In this case, data validity is easier to ensure, and automatic tools are usually available to specify validation rules.

In conclusion, the adopted context data model mainly depends on the supported scenarios and on the aggregation operations to perform. Although almost all the above models offer means to represent QoC metadata, to the best of our knowledge, there are no mature tools to declare and enforce constraints on them at runtime. In fact, the huge design space associated with generic models, as well as the different semantics associated with represented data, do not simplify the definition of completely generic QoC frameworks. Hence, a CDDI usually has to introduce specific solutions to handle metadata for QoC treatment, thus implicitly narrowing the set of context data that can be really handled.

4.2.2. Context Data Storage

Context Data Storage takes care of memorizing data into the distributed CDDI architecture, by also triggering proper coordination and caching/replication protocols if required. As clarified in Section 3.2, the CDDI storage architecture should exploit both physical and logical locality principles to drive context data memorization, so to store data as close as possible to the mobile nodes that will probably require them. From a general viewpoint, context data storage approaches can be categorized in *centralized architectures* and *decentralized architectures.*

Centralized context data storage approaches come with limited scalability and reliability. However, on the bright side, they ease management issues since 1) multiple copies of the same context data cannot exist; and 2) all the context data produced into the system can be easily retrieved by querying the single storage node. Of course, such approaches also simplify the realization of aggregation operators that need access to different context data for the sake of reasoning. Unfortunately, considering the tight limitations on the system scale, very few solutions can adopt this kind of approach.

Moving toward more realistic and decentralized approaches, they make use of different nodes to store context data into the system. In addition, they can adopt either *caching* or *replication* mechanisms that, by carefully storing context data copies into the distributed storage, can increase system scalability and context availability. Although these solutions offer higher scalability and reliability, they lead to increased management overhead; first and foremost, consistency management is an important problem that must be carefully addressed by the CDDI, so to avoid the usage of erroneous or extremely stale data [54]. In the following, we better analyze the introduction and the usage of locality principles to guide both caching and replication mechanisms; for the sake of clarity, we recall that caching mechanisms memorize context data only as consequence of node requests.

Starting with the physical locality principle, it suggests storing context data as close as possible to the associated production point. Since we are dealing with mobile systems, a core CDDI goal is to avoid, if reasonable, mobile nodes spreading context data into the system. Unfortunately, the real-world implementation of such guideline grounds on the provisioning of localization data, either absolute (e.g., obtained by GPS sensors) or relative (e.g., based on fixed anchor points). Localization information must be used to tag context data at production; then, at runtime, context data should be kept as close as

possible to their initial production location, notwithstanding node mobility. This could turn into an extremely hard task according to the adopted network deployment. If the network deployment assumes a fixed infrastructure, and the CDDI can rely on fixed servers to perform context data storage, a simple mapping function between physical places and servers in charge of handling data produced in them completely meets our goal. In fact, the CDDI can automatically store data coming from particular physical places on a predetermined set of servers, and then it can route context requests only to them, thus exploiting the physical locality principle to reduce the runtime overhead. Instead, if the network deployment is a MANET, the potential lack of fixed nodes useful to store the context data greatly complicates the realization of storage architectures guided by the physical locality principle. In the worst-case scenario, namely nodes randomly roaming and proactive replication techniques, the CDDI should continuously migrate context data between mobile devices, thus introducing a high management overhead, also difficult to predict due to the strict dependence with mobility patterns. In all the aforementioned cases, if the CDDI exploits reactive caching techniques, it can only anticipate the removal of context data that are far away from their own production points since, by definition, data transfer only happens as consequence of node requests [57].

As regards the usage of the logical locality principle in context data storage, more challenging issues arise. In fact, the logical locality principle suggests storing the data as close as possible to physical places that exhibit, during particular time hours and days, a skewed interest toward a particular set of context data. Differently from the physical locality principle, here the CDDI has to profile and automatically detect, at runtime, such skewed context data interests, so to detail proper data memorization profiles used by caching/replication techniques. Although the manual definition of such profiles is feasible as well, it does not scale well with system size, and it can require continuous human intervention if context interests dynamically change. In addition, similarly to what we discussed before, storing context data close to a particular physical place is not always straightforward, and strictly depends on the adopted network deployment. In this case, an ad-hoc network deployment introduces also challenging issues for the automatic detection of skewed access patterns, step that usually needs access to the history of the context data requests emitted by multiple nodes.

Finally, let us remark that context data storage is also in charge of handling context data history, namely the possibility of maintaining all relevant past events and retrieving the history of a particular context data. Of course, context data history imposes

requirements on memory resources; depending on data size and on production rates, it could be difficult to maintain the whole history, especially when no fixed servers are available. However, despite required resources, context data history could be fundamental for the correct provisioning of specific services; hence, several real-world CDDIs have to offer such function.

4.2.3. Context Data Processing

Context Data Processing offers all those operations needed to locally shape retrieved context data according to service needs. Usual context data processing covers three main context management aspects: *aggregation*, *filtering*, and *security*. By following this order, we now discuss the possible implementation choices, and we introduce more details about the processing function.

The Context Data Aggregation function provides all the merging operations useful to derive new knowledge from pre-existing context data. Specific operations strictly depend on the adopted context data model and, since context data can be stale and affected by errors, they must be deeply concerned with QoC. The available aggregation techniques can be classified in logic and probabilistic reasoning depending on whether the context data are considered either correct or correct with a specified probability (typically smaller than 1); in addition, hybrid solutions, that combine those two techniques, are also possible. Probabilistic reasoning techniques can usually derive the correctness of composed context data from the correctness of single involved context data. At the current stage, AI provides techniques, and standard logic-based representations and inference engines, that can simplify the usage of aggregation operators; hence, since real-world CDDIs usually require dynamic data aggregation, they adopt either logic- or ontology-based models that are simpler to manage and integrate with those engines.

The Context Data Filtering function strives to increase system scalability by carefully controlling the amount of transmitted context data. These techniques are fundamental since some context aspects change very often, and their associated sources can produce data with extremely high rates. At the same time, context provisioning to services has to be managed according to granted QoC; if services can accept reduced QoC, and that produces less management overhead, context data distribution can use these techniques for the sake of scalability. Filtering operators usually enforce either time-based (new data are not transmitted until a particular time limit is not reached) or change-based constraints (new data are not transmitted if they do not significantly differ from the last transmitted

one). Of course, context-aware services can also define complex filtering operations, made by multiple time-/change-based operators arranged either in parallel or in sequence: for instance, a context-aware service can subscribe for localization change notifications only if the current node position differs from the previous one for, at least, 10 meters (change-based condition), and the last notification has been received more than 10 seconds ago (time-based condition). Notwithstanding the significance of these techniques, they introduce challenging implementation issues that must be properly addressed by the CDDI. In a large-scale network, filters should be not only used locally at the destination node, but also propagated into the distributed architecture to stop the propagation of useless data as soon as possible: the allocation of such filters under multiple criteria, such as minimization of the network traffic, maximization of the sharing of filter operators between different users, and so forth, can lead to very complex optimization problems.

Finally, the Context Data Security function includes all mechanisms to grant privacy, integrity, and availability of data (e.g., to overcome Denial of Service attacks). Real deployment scenarios deeply ask for them because context data could contain sensible information. For instance, while temperature data exchanged in clear text may be not perceived by users as a privacy violation, other context data containing user localization may require appropriate mechanisms to ensure privacy. Although security issues have been already tackled and solved in literature, and efficient solutions to address security problems, e.g., by exploiting access control and encryption mechanisms, are available and usable, we remark that an important part of the privacy loss problem related to the usage of localization data is still open: in particular, indirect inferences of users identity/relations performed on those data represent a real problem that is currently mining the diffusion of these systems [70, 71].

4.3. Context Data Delivery Layer

The Context Data Delivery Layer realizes all the required coordination and dissemination protocols to carry published context data to interested context-aware services. Several solutions are possible, with an important impact on final scalability and context availability. Among different duties, this layer organizes the nodes that take part to the context data distribution, called brokers, to build a particular overlay structure useful to drive both context data and subscriptions routing at runtime. Also, it exploits QoC constraints on the data distribution process to tailor context delivery. Finally, we remark that, of course, the specific context delivery solution must map onto the integrated wireless

communication platform available underneath, and this can limit feasible solutions.

Hence, from a rather general viewpoint, this layer addresses both context data routing schema and overlay structure construction and maintenance. Following this order, now we discuss the main solutions that can be adopted at this layer, with associated pros and cons. For the sake of clarity, Figure 4.3 shows a brief overview of the possible design choices, better discussed in the following subsections.

4.3.1. Context Data Dissemination

Context Data Dissemination enables data flow between sources and sinks. Hence, it is a core function in enabling context access with great impact on context availability and system scalability. A borderline condition is when no dissemination support is needed at all, since sinks directly access sources; we name this category *sensor direct access*. Apart from this strategy, dissemination solutions belong to three different categories, namely *flooding-based*, *selection-based*, and *gossip-based*. The first two categories characterize deterministic approaches where, except during system reconfigurations, a sink always receives matching data produced by sources belonging to the same context data distribution system. Instead, the last category includes probabilistic approaches where a sink can miss some matching data. Systems adopting a hybrid approach that mixes these three main solutions are also possible. Given dissemination crucial role, here we present an in-depth discussion of the associated taxonomy (see Figure 4.3), and we better detail flooding-/selection-/gossip-based categories by introducing additional elements that can help in analyzing real systems.

Sensor direct access approaches may induce low data availability and clash with time/space decoupling because sinks have to communicate directly with sources to access

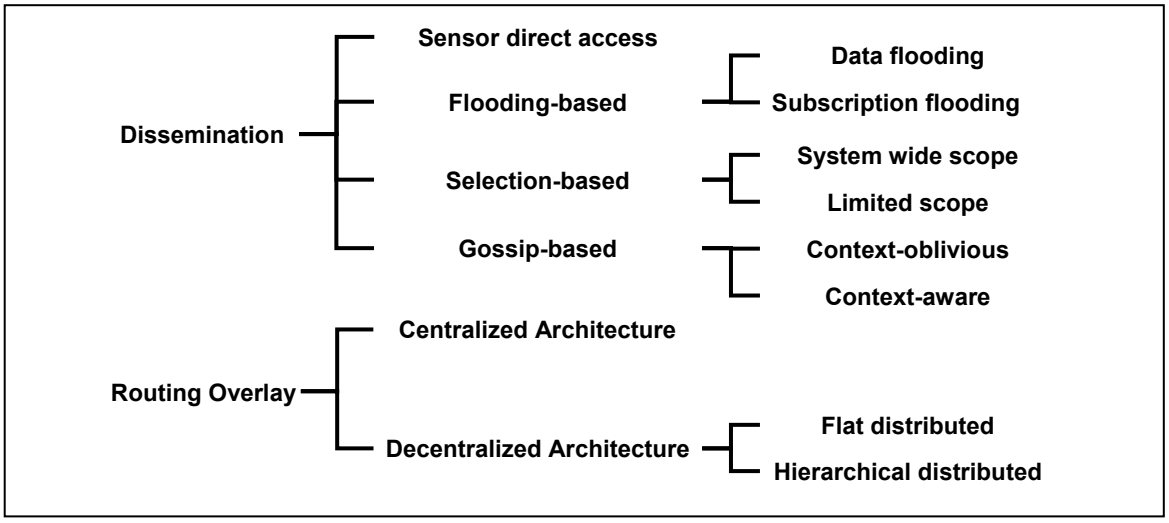

Figure 4.3. Taxonomy for the Classification of the Context Data Delivery Layer.

data; however, as main benefit, they usually result in low complexity. Although seminal works on context-awareness completely relied on this approach, CDDIs for large-scale systems can adopt it only during the initial production phase; after, context data have to be stored and made available into the system according to context-aware service needs, by granting access with no additional constraints on source/sink interactions.

Flooding-based algorithms realize context data dissemination via flooding operations, in other words operations that reach all the nodes contained in a particular scope (such as the entire network, the one-hop neighbourhood in an ad-hoc network, ...). They operate either by flooding context data (data flooding) or by flooding context data subscriptions (subscription flooding). In data flooding, each node broadcasts known context data to spread them inside the entire system, by letting receiver nodes locally select interesting data. Instead, in subscription flooding, each node broadcasts its context data subscriptions to all nodes to build dissemination structure. This schema propagates subscriptions to all network nodes and assumes that each node memorizes subscriptions from all other nodes to perform local matching on produced data. This can reduce bandwidth overhead by disseminating only needed data; however, this solution requires very large routing tables, and that limits scalability.

Selection-based algorithms are typically organized in two phases. In the first one, they deterministically build dissemination backbones by using context data subscriptions; in the second one, data dissemination takes place only over the backbones, and is limited by granting that context data reach only interested nodes. To build backbones, nodes must exchange control information, thus introducing additional management traffic. Selection-based approaches can offer two different visibility scopes to each subscription: system wide scope and limited scope. In the first case, the dissemination process ensures that each subscription is visible in the whole distributed system, so to grant that all the matching data will be retrieved. In the second case, the dissemination process limits subscription visibility to a subset of nodes, for instance the two-hop neighbourhood in an ad-hoc network, so to ensure locality principles and foster scalability; however, due to the limited visibility, it is possible that some matching data will not be found.

Finally, gossip-based algorithms disseminate data in a probabilistic manner by letting each node resend the data to a randomly-selected set of neighbours. Since these approaches do not need complex routing infrastructures to be constructed and maintained, but rather simple local views of the network to choose the neighbours to which gossip data to, gossip-based protocols well fit fast-changing and instable networks, such as MANETs

[72]. It is worth noting that, if correctly tuned, these techniques can ensure high reliability and low latency despite their own simplicity; however, at the same time, they exhibit a runtime behaviour that strictly depends on node density and mobility, and this could lead to unstable performance. We classify gossip-based protocols in context-oblivious and context-aware approaches [73, 74].

Context-oblivious protocols rely on random retransmission probabilities and do not consider any external context information to tailor their behaviour [75]. Between them, pure probabilistic gossip protocols simply resend each received data with a retransmission probability, that can be also different for each node [76, 77]. In counter-based gossip, instead, every time a node receives a new data, it waits a random delay to overhear possible retransmissions by neighbours: at the end of the delay, the node resends the data if and only if it has overheard a number of total retransmissions lower than a threshold [78, 79]. An important finding about context-oblivious approaches is that probabilistic gossip with equal retransmission probability at every node has a threshold behavior: the percentage of nodes that will receive the data suddenly increases when approaching a specific threshold that depends on node density [75, 79]. Hence, main benefit of these approaches is that they involve neither heavy computation nor state on traversed nodes that simply select randomly in the neighbourhood; unfortunately, they can waste wireless bandwidth uselessly by gossiping unneeded data, and do not allow the introduction and the enforcement of quality guarantees due to extremely variable runtime performance.

Context-aware protocols select neighbours by using some external context data potentially belonging to very different context dimensions. For instance, some approaches use physical context (e.g., distance between nodes, local node density, etc.) to position replicas far away [80]; other approaches use social similarity, such as membership to the same university class (user context), to select neighbours to gossip data to. In summary, context-aware approaches reduce the number of useless gossiped data, but they require heavier coordination to exchange and process context data used to make gossip decisions. At the same time, gossip decisions strictly depend on the context data to gossip, and this introduces additional complexities in the CDDI, that has to somehow know the important context aspects to be considered during gossip decisions. Finally, the increased dynamicity makes the runtime behaviour of such protocols not very predictable, thus mining the introduction of quality constraints into the context distribution process.

For the sake of clarity, Table 4.1 briefly summarizes the main characteristics of the dissemination protocols presented before. As main performance indicators, we consider

Table 4.1. Dissemination Protocols Comparison.

Category	Sub-category	Pros	Cons
Sensor Direct Access	-	• No state on mobile nodes • Low network overhead • Sink always receive interesting data • Dissemination reaches only interested nodes	• Strong coupling between sources/sinks
Flooding-based	**Data flooding**	• Low state on mobile nodes • Loose coupling between sources/sinks • Sink always receive interesting data	• High network overhead • Dissemination can reach not-interested nodes
	Subscription flooding	• Loose coupling between sources/sinks • Sink always receive interesting data • Dissemination reaches only interested nodes	• High state on mobile nodes • High network overhead
Selection-based	**System wide scope**	• Loose coupling between sources/sinks • Sink always receive interesting data • Dissemination reaches only interested nodes	• Medium state on mobile nodes • Medium network overhead
	Limited scope	• Loose coupling between sources/sinks • Dissemination reaches only interested nodes	• Sink could miss interesting data • Medium state on mobile nodes • Low network overhead
Gossip-based	**Context-oblivious**	• Low state on mobile nodes • Loose coupling between sources/sinks • Low network overhead	• Sink could miss interesting data • Dissemination can reach not-interested nodes
	Context-aware	• Low network overhead • Loose coupling between sources/sinks	• Medium state on mobile nodes • Sink could miss interesting data • Dissemination can reach not-interested nodes

1) coupling between sources and sinks; 2) state introduced on each mobile node; 3) expected network load; and 4) guarantees on context data delivery. By classifying each dissemination protocol in respect of such indicators, we hope to offer an easy-to-digest overview of available design choices.

4.3.2. Routing Overlay

Routing Overlay takes care of organizing the broker nodes, namely the nodes in charge of real context data routing, into the mobile system. Different architectures can be classified as *centralized* and *decentralized*. The centralized approach includes any possible concentrated deployment (i.e., both single host and clustered), while we classify decentralized architectures into two main subcategories: flat distributed and hierarchical distributed. These latter two architectural choices can help in satisfying the physical

locality principle, for instance, by ensuring that each broker handles only close and easily reachable physical places, and can enhance scalability even if they introduce additional management overhead.

Of course, similarly to what already happened for the dissemination function, the routing overlay approach depends on the adopted network deployment; at the same time, given a particular network deployment, some routing overlay approaches are more suitable according to the adopted dissemination approach. For instance, ad-hoc network deployments are extremely decentralized with possible network partitions and node departures, hence they clash with the realization of centralized overlays. Consequently, in the remainder we consider every single type of routing overlay and, for each one of them, we introduce additional considerations on the adopted network deployment.

The usage of a routing overlay made by a single central broker is appealing due to the guarantees on context data distribution: in fact, due to its centralized nature, the matching process between context data and subscriptions can be efficiently and effectively carried out by contacting the single broker. Unfortunately, this approach comes with low scalability and low reliability, hence, it is suitable only for small-scale deployments, where the context data distribution function serves a small number of sources and sinks. In addition, the feasibility of this approach strictly depends on the adopted network deployment. When fixed wireless infrastructures are used at the network deployment, this approach can be easily supported with a single physical server. In addition, selection-based dissemination protocols with system wide query visibility scope take great advantage from this overlay organization: obtaining system wide query visibility is as simple as routing the context subscription to the unique broker. Instead, when mobile ad-hoc networks are adopted as network deployment, this approach is difficult to apply due to the lack of a static and always available node.

Decentralized approaches, either flat or hierarchical, exploit multiple brokers for the sake of scalability and reliability. They have the advantage that the routing overlay itself can be exploited to enforce locality principles on the context data dissemination. Unfortunately, decentralized routing overlays trade off system scalability and reliability with context availability since the usage of multiple brokers can introduce partial context views; hence, additional management protocols are required to build and maintain a consistent view over available context data. In addition, hierarchical architectures can be preferred to flat ones since they better match the organization of context data with strict physical locality principles, and better drive context subscription routing into the

distributed architecture. Unfortunately, some hierarchical architectures, such as tree-based overlays, can lead to uneven load distribution.

For the sake of clarity, Table 4.2 shows a brief comparison between the possible design choices at the routing overlay. To conclude, the final routing overlay strictly depends on both choices of network deployment and dissemination facility. On the one hand, ad-hoc networks claim for distributed routing overlays (both flat and hierarchical), while fixed wireless infrastructures can exploit all the routing overlay approaches. On the other hand, ad-hoc networks match flooding-/gossip-based approaches since they do not require the maintenance of heavy routing information, while fixed wireless infrastructures prefer selection-based approaches to avoid useless context data distribution and ensure context availability.

4.4. Runtime Adaptation Support Layer

Runtime Adaptation Support enables the dynamic management and tailoring of the other CDDI layers according to current runtime conditions (e.g., available resources, deployment environment, and QoC requirements) with a typical cross-layer perspective. It uses QoC constraints, both on the context data and on the distribution process, to assess the feasibility of possible runtime reconfigurations. For instance, a conflict may arise if the CDDI imposes tight filter operators to reduce exchanged data, and these filters lead to the violation of negotiated QoC. As inappropriate decisions could lead to both system performance and QoC degradation, thus possibly introducing noisy side-effects in context-aware services provisioning, runtime adaptations have to be carefully validated by the CDDI before real enforcement into the system. Finally, although not many solutions have investigated the dynamic adaptation of context data distribution so far, we think this is a

Table 4.2. Routing Overlays Comparison.

Category	Sub-category	Pros	Cons
Centralized Architecture	-	• Context data access is always ensured • No management overhead for routing overlay maintenance	• Limited scalability and reliability • Locality principles difficult to apply
Decentralized Architecture	**Flat distributed architecture**	• Increased scalability and reliability • Locality principles easy to apply	• Context data access could not be always ensured • Additional management overhead for routing overlay maintenance
	Hierarchical distributed architecture	• Increased scalability and reliability • Locality principles easy to apply	• Context data access could not be always ensured • Additional management overhead for routing overlay maintenance

core component in CDDIs for large-scale wireless systems: in fact, it allows the realization of interesting scenarios where the CDDI can adapt context data distribution according to node mobility, available computational resources, QoC objectives, and so forth [3].

Our taxonomy, shown in Figure 4.4, highlights a crucial aspect, namely *service level can affect the adaptation process by influencing the decisions of the runtime adaptation support with different levels of control*. In particular, we classify it as 1) *unaware*, 2) *partially-aware*, and 3) *totally-aware*. In unaware adaptation, the service level neither reaches nor influences runtime adaptation support strategies. In partially-aware adaptation, the service level supplies profiles describing the required kind of service, while the runtime adaptation support modifies CDDI facilities to meet those requests. Finally, in totally-aware adaptation, the runtime adaptation support does not perform any action on its own, while the service level completely drives reconfigurations.

To better clarify how the runtime adaptation support works, Figure 4.5 summarizes the main inputs and reconfiguration policies. The support elaborates both context data inputs (computing, physical, time, and user context) and QoC parameters in order to produce specific reconfiguration commands for both context data management and delivery layers. To be more clear, adaptations can follow five main directions. First, computing context: the runtime adaptation support triggers and executes management functions aimed to overcome changes in the execution environment, such as wireless AP handoff and wireless technology modifications. Second, physical context: the runtime adaptation support modifies data distribution according to physical constraints, such as by exploiting localization information to avoid unneeded data forwarding. Third, time context: the runtime adaptation support modifies data distribution according to specific events or time-of-the-day, for instance, by suspending context data distribution functions during night. Fourth, user context: the runtime adaptation support tailors data distribution to user preferences, for instance, by choosing low-cost connections even if they offer lower bandwidth. Fifth, QoC parameters: the runtime adaptation support dynamically modifies context data dissemination, for instance, by applying proper filtering criteria and differentiated data priorities according to required QoC.

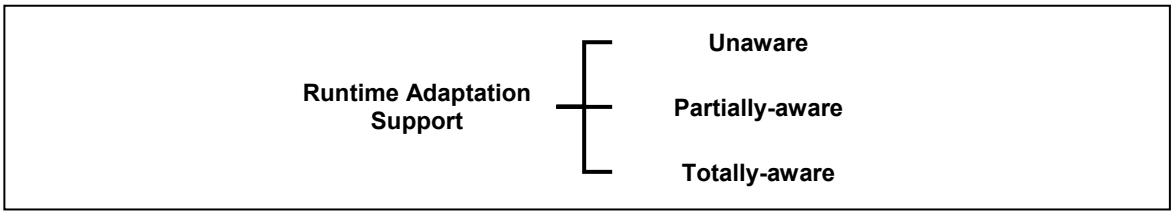

Figure 4.4. Taxonomy for the Classification of the Runtime Adaptation Support.

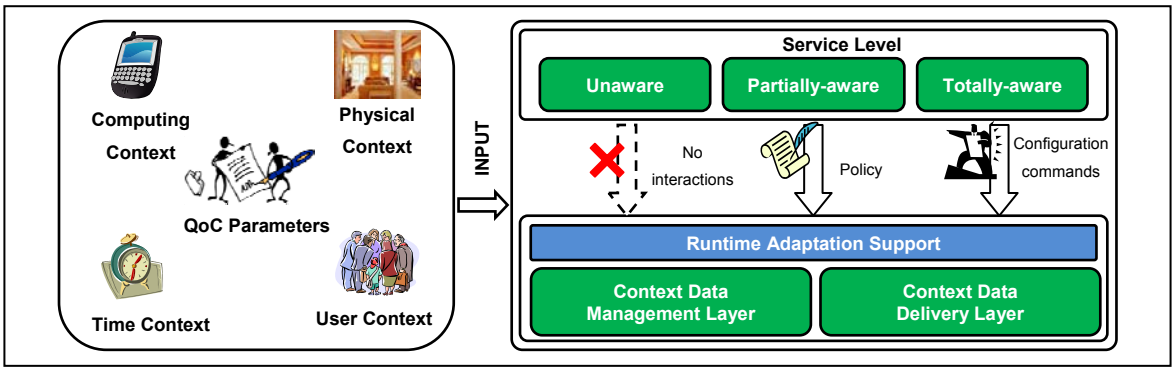

Figure 4.5. Detailed View of the Runtime Adaptation Support.

We remark that the runtime adaptation support should consider all these aspects since reconfigurations can depend from complex conditions, spanning different context aspects and QoC parameters. After the elaboration of these inputs, the runtime adaptation support can command suitable management actions at all different layers. In the following subsections, we highlight the main possible reconfiguration actions performed at each layer.

4.4.1. Context Data Management Layer Adaptation

As regards the context data management layer, common reconfigurations deal with the storage and the filter facility.

Starting from storage reconfigurations, the CDDI can exploit the wealth of context information coming from mobile nodes, namely mobility patterns, current localization information and user profiles, to reconfigure the local storage facilities at each mobile node. In the cooperative context data distribution view, each mobile node shares a local repository of context data with peers; of course, a coordinated management of such repositories is fundamental to increase their own usefulness, by preventing both the memorization of useless data and an excessive number of replicas of the same context data. To be clearer, each mobile node can reconfigure local context data repositories to anticipate the removal of context data that are of scarce interests both for it and for current neighbours; for instance, while roaming, context data far away from their own production points should be removed first. Similarly, if the CDDI dynamically adapts to available resources, it could be the case that a mobile device has to remove many context data due to memory shortage; in this case, an eviction policy, also based on user interests, can be used to keep the elements considered more interesting by the owner of the device.

As regards filtering reconfigurations, they are used to finely tune received context data. QoC notion encourages the introduction of mechanisms suited for data quality, in

particular, filtering operators to enforce quality attributes in suitable ranges. These filters can be adapted at runtime, according to available computational resources both at the mobile node and at the brokers involved into the context data routing. For instance, an interesting adaptation is the runtime merge and split of different routing paths that deliver context data coming from the same source, but with differentiated quality constraints: in this case, the merge of these flows in a unique one, respecting distribution with the tightest QoC constraints, can reduce the runtime traffic of the CDDI. Similarly, since very tight QoC constraints can lead to frequent data exchanges that, in their turn, introduce too much overhead on device resources, the runtime adaptation support could automatically enlarge such QoC constraints to better trade off quality with resource consumption [81].

4.4.2. Context Data Delivery Layer Adaptation

As for context data delivery layer, common reconfigurations mainly adapt dissemination algorithms according to current runtime conditions. Such reconfiguration operations can be extremely varied, spanning from the fine tuning of dissemination protocol parameters, to the dynamic switch of different dissemination algorithms.

Starting with the fine tuning of dissemination protocol parameters, the most interesting solutions are those that realize the context data distribution in a completely decentralized way, mainly flooding-/gossip-based supports for MANET, since those systems need to adapt and to optimize distribution to overcome resource-constrained mobile devices limitations. For instance, gossip-based protocols are usually characterized by two main parameters: 1) a fan-out parameter, describing the number of neighbours that will be hit by a gossip operation; and 2) a gossip period, representing the period of time between two consecutive gossip operations. Data spreading into the mobile network is greatly affected by these parameters, as well as by node mobility patterns. Hence, the runtime adaptation support can reduce the period between two different gossip operations to increase propagation speed for sensible data; instead, it can enlarge it due to bandwidth saturation.

Moving to more complex reconfigurations, the runtime adaptation support can dynamically switch different dissemination algorithms to better fit the current execution environment. For instance, MANETs are usually characterized by mobility patters not easy to predict. Such mobility patterns have a great impact on the effectiveness of the different dissemination protocols, since they can implicitly hinder or favour data propagation into the network; for instance, in data flooding approaches, the more chaotic

the node mobility, the higher the number of different nodes encountered in a limited time span, hence, the higher the propagation speed [82]. Although random mobility patterns simplify data spreading with flooding-/gossip-based dissemination protocols, they make the usage of selection-based protocols almost impossible, due to the high number of routing path reconfigurations and related management overhead. Hence, the runtime adaptation support can dynamically switch different dissemination protocols according to node mobility: if nodes move by building almost stable groups, namely with a low relative mobility, selection-based approaches are feasible; instead, if the relative mobility is high, it could be appropriate to completely change the dissemination algorithm, by favouring the adoption of flooding-based techniques.

4.5. Network Deployments & CDDI Peculiar Aspects

In this dissertation, we focus on the realization of CDDI solutions for large-scale wireless mobile systems. As the adopted network deployment can either favour or hinder context data distribution, here we start by analyzing possible network deployments. Then, since past research works have already addressed the general problem of data distribution, by proposing different solutions to counteract scarce resources and unstable connectivity, we compare our CDDI view with pre-existing data distribution approaches.

Broadly speaking, we consider three categories of network deployments: 1) fixed, that extends the traditional wired Internet by wireless Access Points (APs); 2) ad-hoc, where mobile devices communicate directly with no need of fixed infrastructures; and 3) hybrid, that combines the two previous approaches. In fixed network deployments, the context data distribution function exploits some service reachable through the wireless infrastructure: this grants high context data availability, but also imposes tight constraints on provisioning scenarios as the system is unable to work without infrastructure. In ad-hoc network deployments, the context data distribution function must be implemented in a decentralized way, while ad-hoc links support data transmissions between mobile nodes. These approaches well fit all those scenarios that cannot rely on a fixed wireless infrastructure, but make context availability difficult to ensure; also, all the data management mechanisms need more complex solutions due to possible network partitions. Finally, hybrid network deployment approaches strive to obtain the best from previous ones, with fixed infrastructures that ensure data availability for those nodes able to communicate through them, and ad-hoc communications that may reduce infrastructure overhead and permit to reach nodes unreachable otherwise.

Here we consider three emerging network models, namely Mobile Ad-hoc NETwork (MANET), Vehicular Ad-hoc Network (VANET), and Delay Tolerant Network (DTN), as typical ad-hoc-based network deployments. A MANET is a collection of mobile nodes that use wireless ad-hoc links to communicate; nodes are free to move randomly, thus possibly leading to frequently link breakage and topology changes [44, 45]. A VANET is a MANET whose mobile nodes are vehicles [45, 83, 84]. In these scenarios, nodes have higher speed, but the relative mobility between them is usually low due to limitations introduced by roads and traffic regulations. While MANETs/VANETs usually assume that the path between the source and the destination exists when a message needs to be routed, DTNs accept longer latency and do not assume that the whole source-destination path exists at the same time [38, 85]. A message is routed on a hop-by-hop basis and by following a store-and-forward paradigm, where each node forwards the message to the neighbour that has the highest probability to bring the message close to destination [86].

After this brief presentation of the possible network deployments, we present five main emerging areas very close to context data distribution, namely 1) data distribution facilities for distributed simulations; 2) mobile databases in MANET; 3) multicast and group communication protocols in MANET; 4) pub/sub solutions in mobile environments; and 5) content-centric networking in MANET. Then, we explain why we consider them not suitable to handle context data distribution in large-scale mobile systems.

Starting with brief research area descriptions, distributed simulations need efficient data distribution mechanisms to signal important events between simulated entities that interact among themselves; traditional solutions exploit region-/grid-based approaches to efficiently disseminate events [87]. Mobile database solutions enhance data availability over MANET settings by overcoming possible node disconnections and network partitions. Existing solutions copy data at different mobile nodes by using either caching or replication techniques [56-61]. Multicast and group communication protocols in MANET well fit the context data delivery facility: they allow to create different groups and to distribute data to all the interested entities that have previously joined a group; this model is suitable for distributing context data produced by a context data source to a group of context data sinks. Also, the context data distribution model may seem close to a pure pub/sub model because it is based on sources, sinks, and distribution function [52]. Many solutions for pub/sub in mobile environments have been already proposed in literature, and there exist several context-aware systems that adopt pub/sub systems to perform context distribution [29, 32]. Finally, content-centric networking is emerging as a new

fundamental communication paradigm [88]. Here, contents are univocally identified by an URI used during lookup, and caching at different network stack layers is used to enhance network performance. Different works already proposed content-centric routing over pure P2P MANETs [89, 90], thus assessing the feasibility of this paradigm over such network deployment.

Even if these areas are close to context data distribution, some important differences arise. This section aims to better explain the original need for context data distribution infrastructures for mobile systems; we use our design guidelines to better compare systems belonging to different categories.

Starting with context data production/consumption decoupling, cooperative message (context data) distribution, and adaptation to mobile and heterogeneous environments, these three design guidelines relate to mobile systems in general, hence, they are common to context data distribution and to almost all the research areas presented above. In fact, mobile systems, where nodes freely join and leave the system, make strong coupling between communication entities absolutely unsuitable. Consequently, context data production/consumption decoupling is intrinsic due to the mobile nature of the system, and several solutions belonging to close research fields, such as pub/sub systems, can ensure this requirement [52]. Similarly, the usage of intermediate mobile nodes to cooperatively store messages has been used by several pub/sub implementations for MANETs; in fact, intermediate nodes temporarily store messages and periodically relay them to neighbours, so to cooperatively distribute them. At the same time, the capability of adapting to mobility and heterogeneity is associated with mobile systems in general, because these systems group several mobile devices, spanning from mobile phones and PDAs to full-fledged laptops, with highly different resources; hence, adaptation to heterogeneity is essential and several solutions in the above research areas already support it. At the end, we remark that only data distribution facilities for distributed simulations do not account for adaptation to mobile and heterogeneous environments, as they are mainly used in static and fixed infrastructure environments where powerful servers, connected by high speed links, exchange data to synchronize simulation execution.

If the aforementioned design guidelines are mainly connected with mobile systems in general, and do not allow to clearly differentiate context data distribution from other data distribution approaches, the remaining guidelines carefully suggest that context data distribution, despite some similarities, cannot be fully addressed by other approaches.

Starting with context data life cycle management, all these approaches do not

explicitly handle data life cycle. Mobile databases and pub/sub systems offer seminal solutions to deal with data/message removal, and they do not offer more complex operations, such as data/message aggregation. Similarly, data distribution facilities for distributed simulations and content-centric networks are focused on simple event/content delivery, and do not explicitly consider processing functions. Of course, as long as the system merely delivers data driven by additional and external routing information, the final payload could also adopt complex representation techniques, e.g., first-order logic; however, if the system cannot inspect payloads, different management operations, for instance QoC-based filtering, cannot be implemented. Similarly to QoC-based data filtering, aggregation functions could be obtained by external services running on top the data delivery infrastructure; however, this limits possible operations and final system efficiency.

As regards the enforcement of the context data visibility scopes, data distribution facilities for distributed simulations enforce physical locality into the simulation, e.g., they share events only between entities in the physical proximity, and exploit such principle to optimize the placement of the different simulation components. Mobile database approaches do not usually enforce locality principles, and try to spread data in the whole system to increase availability; this is against the locality principles of the context data distribution. Similarly, both multicast and group communication protocols in MANET and mobile pub/sub architectures strive to build system-wide communication primitives that do not usually enable the enforcement of context data visibility scopes. Of course, differentiated visibility scopes can be mimicked depending on the specific system, e.g., by using the partitioning capabilities usually offered by pub/sub systems to increase overlay scalability [91]; however, these solutions are system-dependent and can lead to increased management overhead. At the same time, it is worth stressing that some pub/sub systems, called *location-aware* in literature [32], can constrain the message/subscription matching depending on the current location, so to enforce limited visibility scopes associated with physical locality principles. Finally, as regards content-centric networks, such approaches dynamically cache and move contents according to the requests currently emitted in the network; hence, they well fit the logical locality principle.

In consideration of QoC-based context data distribution, to the best of our knowledge, all the five research areas highlighted before miss this requirement. Events in distributed simulations do not have any notion of event quality; also, focusing on the delivery process, events are dispatched a soon as possible to prevent the slowdown of running simulation,

with no differentiation in routing delays. Mobile databases do not usually consider quality constraints, neither on the data nor on the distribution process. Even if QoC constraints on context data can be mimicked by local filtering operators, they do not tailor the distributed data delivery process, thus possibly introducing unneeded overhead for replicating out-of-QoC data. In addition, since replication techniques aim to ensure system-wide data consistency, they work effectively only with low change rates, and this is against the fact that context data can rapidly change according to the represented physical phenomenon [56]. Similarly, multicast and group communication protocols tend to ensure consistency between produced and received data: this is against both production/consumption decoupling and QoC-based data filtering. In addition, they strive to deliver data as soon as possible, while leaving out the tailoring of the distributed data distribution process: hence, QoC constraints on the distribution process are usually not supported. Pub/sub solutions do not usually consider quality-based delivery [92]. On the one side, QoC constraints on data can be obtained via message filtering; however, the usage of these filters to tailor the distributed message routing depends on the specific implementation. In addition, context data distribution has to deal with both uncertain data and subscriptions, while the subscriptions made to pub/sub systems consider only perfect subscription/data matches. On the other side, QoC constraints on the distribution process have to be directly supported by the implementation itself as they affect the dispatching process. To the best of our knowledge, previous research on these systems mainly focused on reliable message delivery notwithstanding node mobility, by means of explicit sign-in/sign-off application-level mechanisms and caching proxy servers on the fixed infrastructure [93-96]; these solutions do not consider other quality objectives, such as message delivery time. Finally, content-centric networks enable the distribution of the same content in multiple versions, but they do not usually have any notion of content quality (only content trustworthiness). At the same time, from the delivery process viewpoint, they do not have any means to enforce specific retrieval times and/or priority, thus making impossible the enforcement and the runtime usage of such QoC quality attributes.

To conclude, although context data distribution exhibits some similarities with different research works in literature, none of these approaches fulfills all the context data distribution design guidelines, especially 1) context data life cycle management; 2) locality principles; and 3) QoC-based constraints both on context data and on distribution process. With these observations in mind, we claim that context data distribution for context-aware systems is different from all other traditional data distribution architectures.

4.6. Chapter Conclusions

In this chapter, we presented a logical CDDI model and we investigated the main design choices at each layer. Although it is possible to imagine several in-between solutions, our classification is meant to capture the principal design opportunities. Of course, designers can use hybrid or multiple solutions at a particular CDDI facility to reach different goals, e.g., increasing system scalability or ease of development of local context-aware services. To the best of our knowledge, all the pre-existing context provisioning infrastructures for mobile systems present design choices that can be easily captured by our proposed taxonomies [5]. After, we considered the principal network deployments that can be found in traditional mobile systems. We recalled that general data distribution mechanisms have been already devised and designed in close research areas; although such efforts may seem close to CDDIs, we also explained why we think they do not well fit all the main CDDI requirements. In next chapter, we discuss the three case studies considered in this thesis, and we briefly detail the main contributions presented in the second part of the dissertation.

5. Case Studies

Context data distribution infrastructures for mobile environments are greatly affected by both system size and adopted deployment architecture. On the one hand, as the system grows up to large-scale networks, innovative solutions are required to address both the storage and the distribution of huge amounts of context data. On the other hand, the adopted deployment architecture greatly affects context data availability, as well as context data storage and distribution mechanisms, thus requiring novel solutions for the sake of context provisioning.

The aim of this chapter is twofold. First, in Section 5.1, we introduce three significant case studies of context-aware services for mobile environments, so as to better remark the wide range of different network deployments and quality requirements this thesis addresses. We anticipate that the remainder of the thesis will be organized along these three case studies, since they are good representatives of deployment architectures in real-world scenarios. Then, Section 5.2 better stresses what it is still lacking in today state-of-the-art scenarios, and supplies a brief overview over the main contributions presented in the remainder of this thesis. Let us recall that, although specific solutions are presented in particular scenarios, they can be also applied to other ones.

5.1. Thesis Case Studies

In this section, we detail three significant case studies of very different context-aware services in mobile environments (Table 5.1 offers a brief overview, by highlighting network deployments and main characteristics of each case study). We focus on these scenarios since they feature both extremely different network deployments, ranging from impromptu MANETs to hybrid networks with 3G/4G connectivity, and different quality requirements, spanning from reliable to best-effort context delivery. In addition, each scenario exemplifies a class of context-aware services with a similar network deployment, and represents a set of problems to be addressed and solved in real-world scenarios. By considering these different scenarios all together, one of the goals of this thesis is to reach a better and comprehensive understanding of context data distribution infrastructures for mobile systems.

The first case study regards context-aware services for emergency response scenarios; these scenarios pose several challenging issues due to both unreliable network

Table 5.1. Thesis Case Studies.

Case Study	Network Deployment	Scale	Quality Requirements
Emergency Response Scenario	MANET	Hundreds of nodes	Tight time requirements due to safe-critical services
Smart University Campus Scenario	Wireless infrastructure & MANET	Hundreds of nodes	Best effort delivery
Smart Cities Scenario	Wireless infrastructure & MANET	Thousands of nodes	Different scenarios, from best effort to reliable

deployments and safe critical services. The second case study considers context-aware services for smart university campuses; here, context data distribution can also rely on fixed wireless infrastructures, and context-aware services are usually not safe critical. Finally, the third case study considers extremely large systems by addressing context-aware services for smart cities. This last scenario is also very challenging as it requires both extremely high computational resources, to process all involved context data, and high bandwidth connections, to transmit the context data between the mobile and the fixed infrastructure. In the following subsections, we better detail our case studies; for each one, first we introduce few compelling examples of context-aware services, then we clarify the adopted network deployment.

5.1.1. Emergency Response Scenarios

Disaster area scenarios are usually consequence of an unexpected and sudden disaster, such as earthquake, terroristic attack, etc. In such scenarios, we usually find different rescue teams (doctors, policemen, firemen, …) that coordinate among themselves to ensure a timely and organized reaction. These forces exploit a hierarchical organization in which some leaders tell everybody where, when, and how to work.

The disaster area is usually divided in four principal areas [97]: an incident site, a casualties treatment area, a transport zone, and an hospital zone. The incident site is the area where the disaster actually happened; in this area, rescue teams randomly roam to find and carry injured people to a safe place. Once rescued from the incident site, people are usually brought to the casualties treatment area, a safe area where they receive the first extended medical aid. After this, and only when necessary, people are transported to hospitals. Since hospitals are usually not close to the disaster area, the transport zone contains all those transport units, such as ambulances and helicopters, used to transport injured people.

Between above areas, the incident site is definitely the most challenging and unsafe one: rescue teams randomly walk inside it to find humans in almost unknown place, such

as a building on fire, and do not know the state and the position of injured people. Hence, we will specifically focus on the incident site area, also because here context-aware services can assist the different rescue teams by providing a correct and timely characterization of the current situation. In fact, rescue teams equipped with mobile devices can be assisted by many different context-aware services to 1) automatically obtain rescue operations state, such as the number of still missing people and their positions; 2) distribute involved rescue teams on different sites (floors, houses, etc.) to maximize the coverage area of the rescue operations; 3) prevent the usage of unsafe paths that can put rescue teams on risk; and 4) collect medical records from injured people to anticipate medical needs, such as a particular type of blood for transfusion [98].

Unfortunately, when we consider the real-world implementation of such services, several challenging issues need to be carefully addressed. Starting from the network deployment, these scenarios do not usually assume the existence of fixed wireless infrastructures, as they could have been damaged by the disaster itself. Hence, the final network deployment is usually a MANET, built by the mobile devices carried by rescue team members. Although physical sensors could be deployed either on the mobile devices themselves (e.g., PDAs with temperature sensors) or in pre-existing Wireless Sensor Networks (WSNs) in the incident area, the mobile devices contained into the MANET have to take over all the context distribution responsibilities, spanning from storage to delivery. However, due to the adopted network deployment, intermittent connectivity and network partitions are possible, and can greatly affect context availability. In addition, different quality constraints are fundamental to differentiate high priority from low priority tasks. For instance, consider different rescue teams, e.g., teams of doctors and firemen, trying to gain access to the medical records associated with injured people; while doctors require these data only to monitor people in the casualties treatment area, firemen require them to know if someone still alive is trapped under the rubble in the incident area. Hence, a CDDI for such scenarios has to introduce and enforce differentiated quality levels to favour the routing of data associated with high-priority tasks, such as the ones carried out by firemen.

5.1.2. Smart University Campus Scenarios

Modern university campuses are currently requiring novel context-aware services to enhance students' and professors' life and experience while in the campus. Such services have been already devised in the past, spanning different social aspects and context

dimensions [71, 99]; for the sake of completeness, we now briefly introduce two examples of such context-aware services, namely a context-aware printer and a context-aware event notification service.

As presented in [1], the context-aware printer service is useful to ease interactions between the students and the physical surroundings. As students, especially freshmen, do not know well the main university buildings, this service suggests close printers by melting together context information coming from user location and place profiles describing available physical/logical resources. When a student needs to print a document, e.g., few slides associated with the next classroom, this context-aware service can discover and show all the available printers, ordered by increasing distance from the current user location, so to facilitate and support prompt user decision; after the user opted for one printer, the service automatically and proactively configures the print command to reach the selected printer.

The context-aware event notification, instead, is useful to recommend interesting new events, such as workshops, conferences, seminars, etc., to interested students. The service exploits user mobility, as well as additional information coming from social networks, to refine a user-specific profile that selects interesting events for the user. At the same time, co-location information between users is fundamental to identify social groups, and can enrich user and place profiles to prevent wrong recommendations. When a new event is introduced, the service matches it with user profiles to trigger automatic recommendations. Main goal of this service is to easily promote new events into the university campus, thus fostering wide participation. In addition, this service can help students toward final course exams: it can be used to disseminate study group events, while the service automatically takes over the responsibility of finding interested students.

Both these services can greatly enhance everyday life into the campus. Here, it is feasible to assume the presence of a fixed wireless infrastructure that helps in distributing important context data to final mobile devices. Even more, we can exploit direct ad-hoc links between mobile devices to distribute important context data, thus implementing hybrid network deployments for the sake of context availability and system scalability. Differently from emergency response scenarios, we can assume context data always available through the fixed infrastructure. However, scalability bottlenecks still stand: first, considering that wireless fixed infrastructures deployed in a university campus are usually exploited to provision traditional Internet connectivity, the context provisioning traffic has to be kept as low as possible; second, university campuses present very high

node densities, and this further complicates the context distribution process, that can also require a high amount of bandwidth. At the same time, it is worth noting that context-aware services executed in such environments are not safe critical. Although QoC constraints are useful to manage the distribution process, the CDDI has additional degrees of freedom that can be exploited to finely trade off context quality and distribution overhead. Imperfect and incomplete context views will degrade user experience, but they can be temporarily introduced to deal with overload situations. Hence, by using information coming from both system monitoring and QoC constraints, the CDDI can introduce runtime adaptation mechanisms useful to better limit the introduced management overhead.

5.1.3. Smart Cities Scenarios

In the last years, a new set of city-wide context-aware services is vigorously emerging, thus producing the so-called smart city vision. These scenarios feature city-wide context data sensing and collection, with the main goal of introducing innovative context-aware services meant to reduce city energy consumption, improve citizenship safety, enhance traffic scheduling, and so forth. In this area, we focus on context-aware services for the Bologna downtown: for the sake of clarity, let us briefly introduce few examples.

For the sake of citizenship safety, a context-aware service deployed in Bologna downtown can collect context data coming from multiple sensors and merge them to identify potentially dangerous situations. Video streaming coming from surveillance camera, localization information, and electrocardiogram measurements from local body sensor area networks can be used to detect unsafe situations, such as a person having a sudden heart attack on the street. We expect the smart city to trigger warning messages useful to make persons walking nearby aware of the current dangerous situation: in this case, someone can efficiently intervene to do a first cardiac massage. At the same time, due to the sensitivity of involved tasks, it could be worth triggering ambulance intervention as soon as possible, and alert close paramedics to further increase the possibility of saving a human life.

At the same time, Bologna suffers of traffic jams that can produce long travelling times and high fuel consumption/air pollution. A context-aware service for traffic scheduling regulation can automatically manage these large flows of cars to prevent the gathering of vehicles in the same physical area, so to proactively avoid situations that can lead to traffic jams. In this case, video streaming from cameras deployed on the highways,

as well as cars localization and speed data, can be processed to detect traffic jams formation. After that, the context-aware traffic scheduling service can force cars to follow different paths, by also adapting traffic lights timings to prevent the usage of few roads at all.

Aforementioned examples clearly highlight the great potential of smart cities. However, the real-world implementation of such solutions is very challenging, not only for context data transmission issues, but also for context data processing ones. Although we assume the usage of hybrid network deployments, based on the joint usage of both infrastructure-based and ad-hoc communications, the scale of such system requires novel solutions to correctly handle context data processing and distribution. In fact, smart cities contain thousands of sensors that continuously produce new context data: the storage, the aggregation, and the filtering of such massive amounts of context data cannot be carried out by centralized solutions, hence extremely decentralized solutions are required to address these steps. At the same time, such scenarios present highly variable resource demands, mainly connected to time-of-the-day and location: as people have repeatable patterns during the week, the CDDI can actually predict such resource requirements to plan proper reactions. To effectively handle the last two points, namely large set of context data to be processed and time-varying resource demands, as also stated in Section 2.4.4, we claim the need of Cloud computing architectures [39, 47, 100]. Cloud solutions usually exploit large data centers to offer data crunching facilities, extremely useful to process context data produced by the smart city; at the same time, heavy computations can be carried out by requiring additional hardware/software resources if required. In addition, we have to consider that cellular networks ensure limited bandwidth and may introduce economical costs. This hinders data transfer between the mobile and the fixed infrastructure; hence, the usage of mobile devices to perform initial context data processing is fundamental to avoid the introduction of high traffic from/to fixed wireless infrastructures. Finally, focusing on QoC management, smart cities feature several services that can span from safe critical ones, such as accident prevention and citizenship safety monitoring, to completely best-effort ones, such as automatic recommendation of interesting events in the physical surroundings. Hence, a CDDI for such scenarios has to provide and enforce a wide set of QoC requirements, by also carefully handling the interactions with heterogeneous wireless networks that could be not under the direct control of the CDDI itself.

5.2. Intermediate Conclusions & Contribution Outline

In view of the first chapters, we can now better remark the main goals of this thesis work. In Section 3.4, we introduced a selection of context-aware systems currently available in literature. These valid works addressed different and heterogeneous issues, spanning from local context provisioning mechanisms to principles and architectures for context management. However, to the best of our knowledge, they did not consider the real-world implementation of such scenarios in large-scale deployments as a first objective. Heterogeneous wireless networks, as well as wireless modes, introduce interesting opportunities and issues; above all, hybrid network deployments suggest cooperative context distribution schemas where part of the distribution process and load is offloaded to and operated by the mobile network through ad-hoc links. Apart from the effects induced by the network deployment on the CDDI, we also think that quality-based context provisioning has been widely neglected in the past. Although one of the oldest work on QoC was published in 2003 [3], since then only few works in literature, such as SOLAR [33], considered QoC constraints as fundamental drivers to manage the distribution process. Hence, we feel that additional research work is required to effectively consider QoC constraints at runtime, so to also self-adapt the CDDI according to available resources.

We conclude this chapter by anticipating and highlighting the main contributions of this thesis along our three case studies. In a very synthetic overview, since emergency response scenarios stress the problem of context data availability and retrieval in completely decentralized networks, in Chapter 6, we present 1) a *quality-based context data caching* approach, which uses quality constraints to dynamically reconfigure caching facilities; and 2) an *adaptive query flooding* approach, that enforces maximum data retrieval time, while reducing the number of exchanged messages. Moving to smart university campus scenarios, in Chapter 7, we extend our context data routing protocols in hybrid deployments and we introduce 1) an *adaptive context data caching* approach, that strives to detect current access patterns to adapt the ranking function; 2) an *adaptive data/query batching* approach, which exploits delay tolerance to enable batching solutions; and 3) an *adaptive query drop* approach, that dynamically adapts the number of processed queries to limit the introduced CPU load. Finally, in Chapter 8, we focus on the integration of Cloud solutions to perform context data storage and processing: we present a new *network-aware VM placement algorithm for Cloud systems*, whose main goal is to

increase system stability under time-varying traffic demands, for instance, consequence of context data flows that dynamically change due to a sudden gathering of people in a particular physical area. For the sake of clarity, Table 5.2 briefly summarizes our main contributions for each case study.

Table 5.2. Outline of Practical Thesis Contributions.

Case Study	Main Practical Contributions
Emergency Response Scenario	Quality-based Context Data Caching Adaptive Query Flooding
Smart University Campus Scenario	Adaptive Context Data Caching Adaptive Data/Query Batching Adaptive Query Drop Policy
Smart Cities Scenario	Network-aware VM Placement for Cloud Systems

Finally, let us recall that this chapter ends the theoretical part of this thesis work. The following chapters focus on the practical contributions of our work, by presenting algorithms and protocols design, as well as extensive experimental results for the sake of performance evaluation.

6. Context Data Distribution in Emergency Response Scenarios

This chapter focuses on the realization of context-aware services in disaster area scenarios. In Section 6.1, we present an in-depth discussion of the main issues, as well as of the design guidelines, adopted by our CDDI. After a brief introduction of the distributed architecture, given in Section 6.2, we follow the logical CDDI architecture presented in Section 4.1 and we introduce the main solutions adopted by RECOWER at each logical layer in Section 6.3, Section 6.4, and Section 6.5. Above all, we focus on the usage of differentiated QoC levels to reconfigure both context data storage and delivery. Finally, Section 6.6 presents implementation details of RECOWER, while Section 6.7 discusses experimental results, obtained through simulations, showing that self-adaptive mechanisms, guided and constrained by required QoC levels, can effectively and efficiently optimize the data distribution.

6.1. RECOWER CDDI

Context-aware services in disaster area scenarios are extremely significant [98]. However, the realization of real-world CDDIs for such scenarios presents still open and challenging issues, associated with both the delivery and the storage of context data. In fact, these systems need high bandwidth and reliable wireless links, all properties that clash with traditional bandwidth-constrained and unreliable ad-hoc wireless technologies; also, since all devices are usually located in the same physical area, they form a local wireless network where transmission collisions are usual rather than unexpected. In addition, these systems have to handle huge amounts of context data by using a distributed data repository based on available mobile devices. As MANETs are likely to lead to network partitions, the system has to explicitly deal with context data availability, by introducing proper caching/replication techniques. In conclusion, since emergency response scenarios usually exploit a MANET as network deployment, we claim the significance of the following main design guidelines.

First, such a CDDI should integrate with and use any wireless technology available to injured people devices. Among different technologies, a CDDI for disaster area scenarios should support, at least, both WiFi and BT, as the most widespread ad-hoc wireless technologies. In this way, the CDDI can increase both the final available bandwidth, by improving system scalability, and the communication opportunities, by reducing the

probability of network partitions. Toward the integration of heterogeneous wireless technologies, the CDDI should adapt both context distribution and management protocols depending on available resources; for instance, it should automatically reduce context data delivery rates to avoid wireless channel saturation.

Second, the CDDI should realize a distributed data repository. Since traditional mobile devices usually have limited storage capabilities, we cannot assume a centralized solution in which one node collects and supplies access to all the data available into the disaster area. Consequently, the context data must be spread over the whole system and, at the same time, data caching/replication techniques should take place to increase data availability and to reduce the average path length required to access context data [56].

Third, the CDDI should exploit both the physical and the logical locality principles to carefully store context data copies into the system. Both these principles are useful to optimize available resources, but they must be carefully applied since they reduce the number of data copies, thus potentially leading to reduced context availability. In addition, from an implementation viewpoint, the CDDI should exploit localization data to impose geographical distribution bounds. When a localization support is not available, e.g., when the mobile devices are in indoor disaster areas, the CDDI should be able to exploit other techniques, for instance based on hop count, to reduce distribution scopes.

Finally, the CDDI should be both QoC-based and context-aware in itself. Since we have limited resources and an amount of context data that could saturate the whole bandwidth, the CDDI has to introduce, enforce, and use QoC constraints to differentiate context data storage and delivery. For instance, to ensure QoC data retrieval time, i.e., the time period between context request and real data delivery to the mobile node, the CDDI should dynamically adapt to distribute first the data closer to delivery deadline. Also, since these environments are densely populated, with several devices using the same wireless channel, the CDDI can exploit context-awareness to optimize the distribution process. For instance, as long as the QoC data retrieval time is ensured, each node can introduce routing delays to coordinate with neighbours and to understand which data have been already distributed; in such cases, the CDDI can prevent further distributions, thus avoiding the introduction of useless overhead. Additionally, data caching/replication techniques should be managed to increase data diversity between near nodes, so to keep local more data and to increase the probability of retrieving required data in near peers. This also results in reduced context data retrieval times since mobile nodes have higher chances of finding required context data on close neighbours.

Following these guidelines, we designed our CDDI for disaster area scenarios, namely Reliable and Efficient COntext-aware data dissemination middleWare for Emergency Response (RECOWER) [101]. In the remainder, after a brief introduction of the RECOWER distributed architecture, we detail the fundamental mechanisms and solutions adopted at the different CDDI layers to improve distribution scalability and reliability.

6.2. A Proposed Distributed Architecture

RECOWER adopts a simple distributed architecture due to the constraints imposed by the deployment. We remove any assumption about the existence of fixed wireless infrastructures; hence, the RECOWER distributed architecture is a MANET built by mobile devices carried by rescue team members. WSNs can be used during the context data production phase but, since they offer short communication ranges and limited battery, important context data (e.g., temperature readings and localization data associated with people trapped under the rubble) are always offloaded to mobile devices. In this way, RECOWER CDDI has full control over such data, and can either cache or replicate them for the sake of context availability. For the sake of clarity, Figure 6.1 shows an example of the traditional distributed architecture adopted by RECOWER.

In our vision, each mobile node executes a local RECOWER instance, and hosts multiple and heterogeneous wireless interfaces. Hence, our CDDI uses a Peer-to-Peer (P2P) heterogeneous MANET that integrates both WiFi and BT. To enforce the physical locality principle, while avoiding the strong assumption of localization data provisioning, RECOWER does not exploit MANET multi-hop routing protocols at the network layer,

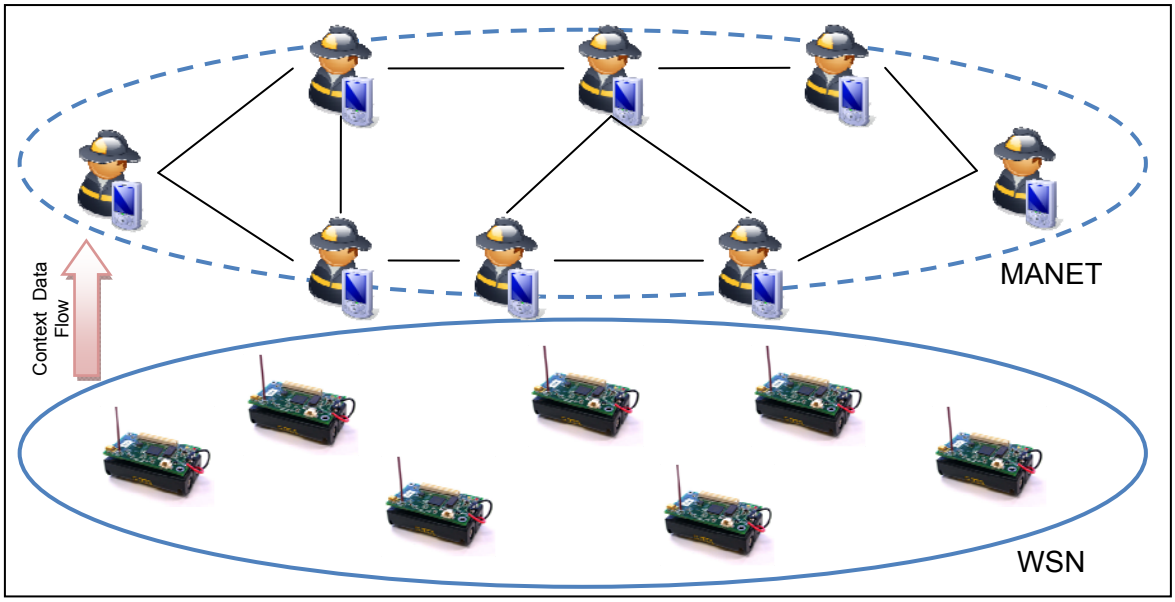

Figure 6.1. Example of a Traditional RECOWER Deployment.

and manages data routing at the application layer on a hop-by-hop basis between neighbours, in order to have a rough estimation of the distance. Finally, as most of the routing protocols for MANET do [102], RECOWER assumes a beaconing mechanism to handle node mobility: in other words, each mobile node periodically emits a broadcast beacon to signal its presence to its own one-hop neighbours.

6.3. Context Data Management Layer

RECOWER addresses context data distribution in completely decentralized MANETs. Of course, this introduces challenging issues for the realization of both context data storage and processing facilities, since node mobility can lead to remarkable changes in system settings. This makes the implementation of distributed algorithms and coordination protocols very hard, as each mobile node can experience high variation rates in its own one-hop neighbourhood. Due to the adopted network deployment, RECOWER context data management uses highly localized solutions to ensure context data provisioning to service level with timeliness constraints and high reliability.

Starting with context data representation, RECOWER adopts an object-oriented approach, where each context data instance is an object that offers access to its own attribute values through proper methods. Apart from the attributes describing real context aspects, each context data instance is associated with additional management parameters. *Source ID (SID)* is the unique identifier associated with the context source that produced this data. *Version Number (VN)* is an increasing number, attached by the source, used by mobile nodes to distinguish older data instances from newer ones. *Foreseen Lifetime (FL)* is the maximum data lifetime estimated by the source at generation, while *Remaining Lifetime (RL)*, initially equal to FL, is dynamically decremented to account for time elapsing; when RL is zero, the data is no longer valid, and it is removed by context data repositories. Finally, to enable QoC-based data management, RECOWER tags each context data with proper quality metadata. In the remainder, we assume that each data instance has a QoC up-to-dateness parameter equal to the ratio between RL and FL (hence, in [0; 1]), useful to express the probability that the associated data instance is the latest one produced by the source.

As regards context data storage, RECOWER mobile devices memorize all the data produced by dynamically discovered WSNs (see Figure 6.1); at the same time, also on-board sensors produce new context data to be shared into the system. The memorization overhead is not trivial to be addressed due to resource scarcity, and the realization of

history mechanisms, in charge of providing historical values of context data, introduces additional issues due to both storage requirement and clock synchronization. On the one hand, resource consumption is not a fundamental objective when human lives are at stake; hence, although context data storage is limited by device capabilities, RECOWER does not strive to optimize the usage of memorization resources. On the other hand, instead, context availability is fundamental since some data can contain safe critical information, e.g., localization and temperature information of the incident area. Hence, RECOWER adopts caching mechanisms to store multiple copies of the same data, in order to increase final context availability. In the remainder, each RECOWER node has a limited data repository, with maximum size D_{MAX}, that stores context data instances, either locally produced or received by remote nodes in response to locally issued queries. When the repository is full, we evict the Least Recently Used (LRU) element.

Moving to the context data processing facility, it is completely implemented by means of local solutions. RECOWER CDDI offers context data aggregation and filtering operators, but all the associated algorithms are executed in a local fashion by assuming to have needed context data available in the local storage. This introduces an increased overhead on the mobile device, but it is feasible for the following main reasons. First, as stated before, in emergency response scenarios both battery draining and CPU/memory usage are not important objectives to deal with. Second, distributed context processing solutions are unfeasible due to the fast changing network conditions: they will probably result in high network overhead and, at the same time, both the convergence and the reliability of such processing operators are difficult to ensure. Since RECOWER has to provide context data to the service level as soon as possible, it is better to locally elaborate them, rather than to wait the convergence of external routing protocols and distributed mechanisms. Due to the introduced overhead, that design choice limits the maximum number of processing operators executed, at the same time, on the mobile device; hence, it trades off scalability and reliability of executed context processing operators.

Finally, let us remark that context data confidentiality, integrity, and availability are fundamental in such scenarios due to the safe-critical nature of executed context-aware services. We did not consider the security issues introduced by RECOWER CDDI since out-of-scope in respect of the main objectives of this thesis work. However, we remark that a real-world implementation of such CDDI definitely needs security mechanisms to avoid malicious users inject wrong context data that could lead to extremely dangerous situations, e.g., by hijacking rescue teams along unsafe paths. Interested readers are

referred to [103] for an in-depth investigation of trust management schemas in MANETs.

6.4. Context Data Delivery Layer

Similarly to what happened for the context data management layer, RECOWER context data delivery layer is also widely affected by the adopted network deployment. The fast changing nature of MANETs does not fit well the building and the maintenance of complex routing infrastructures that, although suitable for more static scenarios, can lead here to excessive management overhead and unstable runtime performance.

In further details, at the dissemination facility, we adopted a subscription flooding approach to prevent the distribution of context data not explicitly required by mobile nodes. Hence, when a context-aware service requires particular context data, RECOWER builds a subscription, i.e., a query in the remainder, and distributes it to neighbours. At the routing overlay facility, we used a decentralized and flat solution to avoid the introduction of additional management overhead due to cluster formation and maintenance [104]. Since each mobile node memorizes and shares local context data with neighbours, the delivery layer distributes context queries to all the current nodes into the one-hop vicinity.

RECOWER context routing is based on two main entities, namely *context data* and *context queries*. While the former ones represent the real context information, the latter ones are used to build temporary distribution paths that drive context data routing into the distributed architecture. We recall that each context data has always, at least, its up-to-date version memorized at the creator node to ensure availability; additional distributions happen only if matching queries exist, otherwise data will be not distributed. Each query carries a *data filter* used to select matching data depending on context requirements; in particular, the data filter is specified by the sink at the query creator node, and is made by a set of simple constraints on context data attributes (e.g., membership conditions, range conditions, etc.) arranged through AND/OR functions. In addition, RECOWER context data distribution follows two main management directions. First, it considers both data up-to-dateness and retrieval time to adapt data/query distribution; while the former is used to tailor data/query matching, the latter is useful to modify routing delays, for instance, to favour the routing of important data close to retrieval time expiration. Second, it controls employed resources and runtime overhead; query replication increases reliability, but replication degree and number of transmissions have to be carefully managed to avoid the introduction of scalability bottlenecks. Now, we present main management parameters, as well as mapping processes, used by RECOWER to reach these high-level objectives.

Starting with query transmission, we remark that RECOWER adopts a one-hop broadcast-based approach to distribute context queries. In this way, each node can distribute a query to its own entire neighbourhood with the lowest possible transmission overhead. Data distribution, instead, adopts a unicast-based approach where each node sends data only to the node that had relayed the query for the following two main reasons. First, if a node broadcasts a context data instance, that could trigger caching mechanisms and replacement policies on all the reachable neighbours: as consequence, this can both reduce cache diversity and introduce trashing behaviour. Second, this transmission policy permits to better control the number of data transmissions: since context queries are distributed in broadcast, more nodes in the same physical area could store a particular context query, hence data broadcasting could trigger a very high number of retransmissions.

For the sake of clarity, Figure 6.2 shows a context data distribution example. Two nodes linked by a continuous line can communicate directly since into the transmission range of each other. In Figure 6.2 (a), A starts distributing a query Q_A: the first transmission hits its neighbours, i.e., B, C, and D. By assuming that B, C, and D do not have a positive match for Q_A, all of them schedule a new query distribution after a random delay lower than a fixed *Query Routing Delay (QRD)*. As clarified in the following, this random delay lets each node monitor query transmissions and self-adapt according to current network load conditions. After another query distribution round (Figure 6.2 (b)), all the nodes have a stored copy of query Q_A (Figure 6.2 (c)). Assuming that E has some data matching Q_A, it schedules a data distribution after a random delay less than a fixed *Data Routing Delay (DRD)*. Differently from QRD, this delay strives to prevent wireless

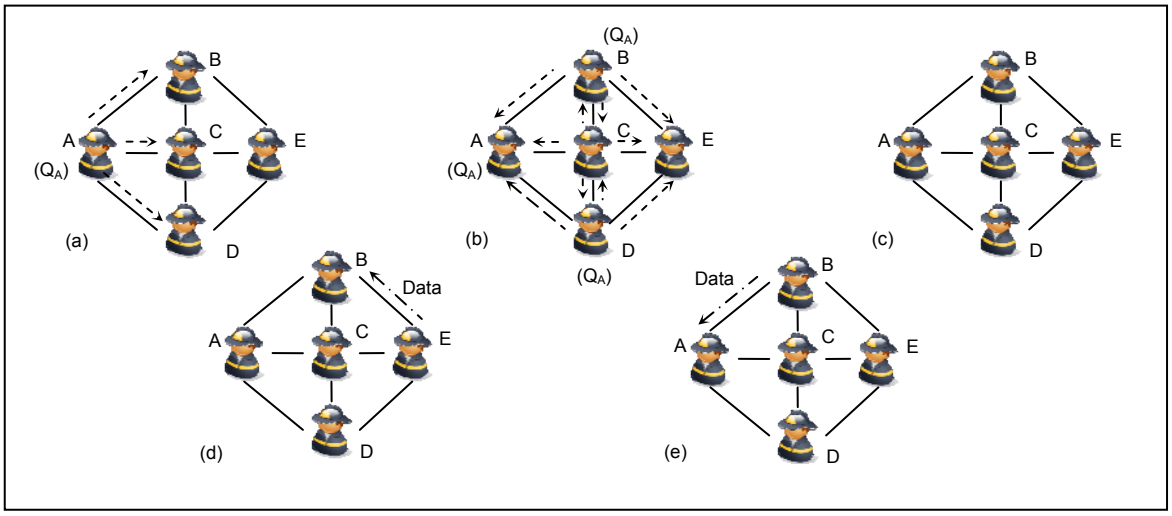

Figure 6.2. RECOWER Context Data Distribution Process.

collisions between queries and data, so as to avoid the well-know wireless storm problem [105]. Finally, E relays the data to the node from which it received Q_A the first time (i.e., B in Figure 6.2 (d)), that, in its turn, sends them to the previous node in the query distribution path (i.e., A in Figure 6.2 (e)) after another random delay lower than DRD.

In RECOWER, each context-aware service supplies proper QoC constraints in order to drive context data distribution. Focusing on the context data delivery layer, QoC data retrieval time is fundamental to properly set the routing delays presented in Figure 6.2. Also, an automatic mapping function is required to obtain low-level query parameters from the high-level QoC data retrieval time. Before we detail the mapping process, let us recall that, to enforce QoC constraints and manage query aging, each RECOWER query has six main management parameters. *Time To Live (TTL)* is the maximum number of hops a query can traverse; it is decremented at each traversed node, and does not allow further distributions when zero. *Maximum Query Response (MQR)* is the maximum number of data instances collected by this query; it is used to prevent excessive data distributions, for instance, when we look for context data that could have been produced by several sources in the same physical area. *QRD* and *DRD*, as introduced above, are the two maximum delays each node can introduce respectively during query/data distribution. *Already Collected Data (ACD)* is the list of the keys associated with the already routed data, and is fundamental to prevent retransmissions of already collected data instances. Finally, *Query LifeTime (QLT)* expresses a deadline after which the query is expired and removed by the system.

As regards the mapping between the high-level QoC data retrieval and the low-level query parameters, we consider the following process. Query TTL is a service-level parameter that depends on the required distribution scope. QLT is equal to the data retrieval time; since all the data routed with more delay than data retrieval time are out-of-QoC, RECOWER removes associated queries to avoid unneeded overhead. Finally, once selected a proper TTL, we consider that each node in the distribution path introduces a maximum delay of DRD in data distribution, and a maximum delay of QRD in query distribution. Hence, the ratio between the data retrieval time and the query TTL is the maximum delay each node can introduce. This value is equally split between DRD and QRD, and the final random data (respectively, query) delay applied during distribution is selected by a uniform distribution in the range $[\alpha; \beta] \times$ DRD (respectively, QRD), with $0 \leq \alpha < \beta \leq 1$.

Finally, let us remind that RECOWER manages also memorization resources

employed by context queries. Even if each node can potentially store a large amount of context queries, RECOWER aims to limit them to avoid overload situations: in fact, the memorization of many context queries can trigger several data distributions that, in their turn, contribute to increase wireless network load. Consequently, each RECOWER instance has a limited query repository, with a maximum size of Q_{MAX}, to store queries received by neighbours. If necessary, a replacement policy is used to identify the query to evict; here, the query replacement policy evicts the query closest to its own QLT, since it is the one that has more chance of not respecting its associated deadline.

6.5. Runtime Adaptation Support

RECOWER exploits QoC constraints to optimize the main mechanisms introduced in Section 6.3 and Section 6.4. According to our classification presented in Section 4.4, we adopted a partially-aware approach: each context-aware service provides high-level QoC constraints, while RECOWER reconfigures associated distribution mechanisms to optimize runtime performance. In particular, to increase both context data availability and distribution reliability, RECOWER adapts 1) context data replacement at the context data management layer; and 2) queries distribution at the context data delivery layer. Nevertheless, to ease the development of context-aware services, RECOWER defines a standard set of *differentiated quality classes*. Each Quality Class (QC) introduces 1) quality constraints on received context data; and 2) maximum data retrieval time during routing. The QC of each service is statically defined at deployment time according to its safe criticality.

In the remainder, we present the runtime adaptation support of RECOWER. Since this is a vertical module that crosscuts different aspects (see Figure 2.1), this section is organized in two main subsections: the first one, Section 6.5.1, introduces how RECOWER adapts context data caching according to QoC requirements; then, the second one, Section 6.5.2, introduces how it adapts context query distribution to reduce the number of distributed messages.

6.5.1. Adaptive QoC-based Context Data Caching

At deployment time, each context-aware service is associated with a QC that defines quality constraints on received context data; our CDDI uses them at runtime to tailor the context data/query matching process, so to route only data that will be used by receiving sinks. For the sake of management, quality classes are arranged in a hierarchy that defines

containment relations between higher and lower quality classes. If $M(QC_i)$ is the set of data accepted by a user belonging to QC_i, and QC_i is a quality class higher than QC_j, the relation $M(QC_i) \subseteq M(QC_j)$ has to be always true. For instance, firemen and doctors have two different quality classes, respectively QC_1 and QC_2. Firemen need context data with a up-to-dateness parameter higher than 0.7, while doctors accept data with every possible up-to-dateness ($M(QC_1) \subseteq M(QC_2)$). All the produced context data have an initial up-to-dateness value of 1.0; decreasing values are assigned while time passes according to a particular function, for instance, a linear function. Hence, at the beginning, all context data match both firemen and doctors quality constraints; instead, when the up-to-dateness value decreases to less than 0.7 due to time elapsing, the context data will no longer match firemen quality constraints.

As the usefulness of caching a context data instance degrades when its quality attributes tend to be out-of-QoC for all the nodes in physical proximity, RECOWER exploits quality classes to maintain only the data with probability of being reclaimed in the future. It considers that nodes belonging to low quality classes, namely quality classes with loose constraints, can cache context data with very poor quality. Those cached data are completely useless if the node caching them is surrounded by mobile nodes with higher quality classes; in fact, all the queries will not find positive match due to poor data quality. At the end, such situation wastes precious storage resources; hence, RECOWER strives to anticipate the removal of low quality data, in order to keep context data that, according to their own quality attributes, can be required in the future by close neighbours.

To conclude, QoC-based context data caching has to consider quality classes of close physical neighbours. The data repositories hosted on mobile nodes with low quality classes have to be dynamically adjusted according to the current neighbourhood; in particular, if required, RECOWER will adapt data caches to store context data with high quality attributes. The specific mechanisms used by our CDDI to select the quality-based constraints on data, as well as implementation details, will be presented in Section 6.6.2.

6.5.2. Adaptive Context Query Flooding

Context query flooding is an expensive phase that, apart from requiring high network resources, deeply affects distribution process reliability; in fact, broadcast messages in WiFi networks are less reliable than unicast ones due to the absence of the RTS/CTS mechanism. Hence, the context query distribution phase adopted by RECOWER has been optimized following two main directions, so as to avoid useless query distributions and to

limit excessive routing overhead. Let us give some concrete examples. First, in Figure 6.2 (b), two of the three distributions performed by node B, C, and D are useless; since E is the only one that had not already received the query, one of these distributions is enough to ensure query distribution coverage. Hence, RECOWER aims to avoid useless query distributions that will hit only nodes that had already received the associated query. Second, although broadcast-based context query distribution is appealing due to distribution paths replication and reliability, it wastes a lot of network resources and can easily saturate the memory available for query memorization on mobile devices (limited by the Q_{MAX} parameter). This, in its turn, will result in query replacement and consequent routing path breaks. To avoid such problems, RECOWER tailors the broadcast-based query distribution to hit only a subset of the current neighbours.

With a finer degree of detail, to avoid useless query distributions, RECOWER tries to identify whether all the current neighbours had already received the query. Toward that goal, each query has an *Already Disseminated Nodes List (ADNL)* parameter that contains all the identifiers associated with the nodes that had already received the query (in the remainder, Q_{ADNL} is the ADNL parameter of the query Q).

For the sake of clarity, Figure 6.3 shows a context data distribution example to clarify how our solution can prevent useless query distributions. At runtime, each node periodically emits a beacon message to signal its presence to its own one-hop neighbours (Figure 6.3 (a)). Hence, each node has a local Routing Table (RT) containing all the communication end-points associated with current one-hop neighbours. Of course, depending on both beaconing periods and node mobility, some inconsistencies can arise: a node can have both incomplete view (e.g., a node that is just arrived and that had not emitted any beacon yet) and not up-to-date view (e.g., a node that is not into the transmission range anymore) of the current neighbourhood.

When A wants to distribute the query Q (Figure 6.3 (b)) with a TTL of 2, it computes the difference between the current list of neighbours (contained in RT_A) and the Q_{ADNL}. Q_{ADNL} is initially empty, hence, the difference is {B, C, D}, meaning that some neighbours had not already received the query. Node A adjusts the ADNL to contain {A, B, C, D}, and then broadcasts Q. After having received the query, B, C, and D try to match it with locally stored data. Assuming that none of them can supply a response, and since the associated query TTL is higher than 0, they schedule a next distribution after a random delay lower than the associated QRD. If C is the first node that distributes Q again (Figure 6.3 (c)), it calculates the difference between RT_C and Q_{ADNL} (RT_C / Q_{ADNL} = {E}), updates

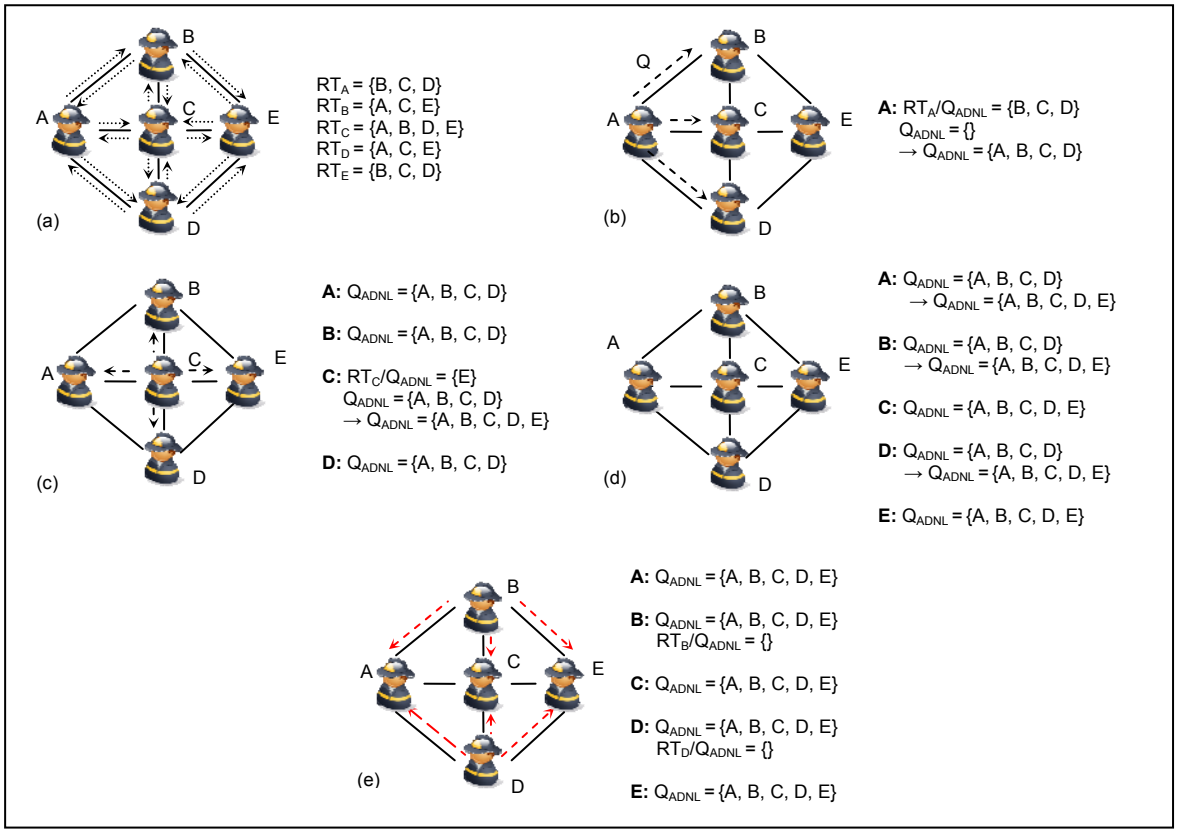

Figure 6.3. RECOWER Adaptive Query Flooding.

the ADNL by appending E (Q_{ADNL} = {A, B, C, D, E}), and broadcasts the query. This query distribution will hit A, B, D, and E since into the transmission range of C. As nodes A, B, and D already have a local copy of Q, they simply update the local copy of the query ADNL by appending the ADNL carried into the received query (Figure 6.3 (d)); in this way, they try to reach a potentially up-to-date vision of the nodes that had already received the query. Instead, E receives the query for the first time, and tries to perform data matching. When B and D (Figure 6.3 (e)) try to distribute Q again, the difference between the associated RT and the query ADNL is empty; hence, all the current neighbours had already received the query, and the two query distributions are suppressed (see red arrows in Figure 6.3 (e)).

Instead, to control query replication in the same physical neighbourhood, RECOWER reduces the query distribution scope by imposing that a single broadcast message is actually processed by only a subset of the current neighbours. In other words, by always using a single broadcast query distribution, our CDDI imposes which neighbours have to consider the received query Q. Toward this goal, RECOWER exploits the query ADNL parameter as well. When a query Q has to be distributed, the sender node selects the set of neighbours to which the query will be actually sent and, before the real transmission, inserts their identifiers into Q_{ADNL}. When a node receives Q, it checks that its own

identifier is into the Q_{ADNL}. If the test is positive, the receiving node considers the query; otherwise, it discharges the query since it is not into the set of neighbours selected by the sender node. At the same time, even if the query is not to be processed by the receiving node, we 1) update the local query ADNL (if available) to improve the efficiency of the query distribution suppression; and 2) match it with locally memorized data to spread them and to increase context availability.

Figure 6.4 shows a simple example. Node A has to distribute Q with a TTL of 2. Using its local RT_A, it detects three neighbours that had not already received the query, namely B, C, and D. Before distributing the query, it uses a selection process (detailed in Section 6.6.3), and decides to distribute the query only to B and C. It updates Q_{ADNL} by inserting its own identifier and {B, C}, and then broadcasts Q (see Figure 6.4 (a)). Due to the broadcast nature of the wireless channel, B, C, and D receive Q (see gray circles in Figure 6.4 (b)), and try to match Q with locally memorized data. After this phase, each node tests if its own identifier is into the Q_{ADNL}. Due to test results, only B and C (dotted circles in Figure 6.4 (b)) consider the query, and schedule further distributions due to the TTL higher than 0, while D silently drops the query and does not perform any distribution.

To conclude, by using query ADNL parameter, each sender node can control which neighbours will actually consider a broadcast query. Of course, this solution introduces computational overhead on all the one-hop neighbours (due to data/query match and ADNL test), but lets us to select and distribute a query only to a subset of the physical neighbours by using a unique broadcast transmission.

6.6. Implementation Details

This section introduces the main implementation details of RECOWER. We anticipate that RECOWER has been fully implemented on a network simulator, namely NS2, to better study the effects of our policies in large-scale mobile systems. Section 6.6.1 introduces the software architecture of our CDDI; then, from Section 6.6.2 to Section

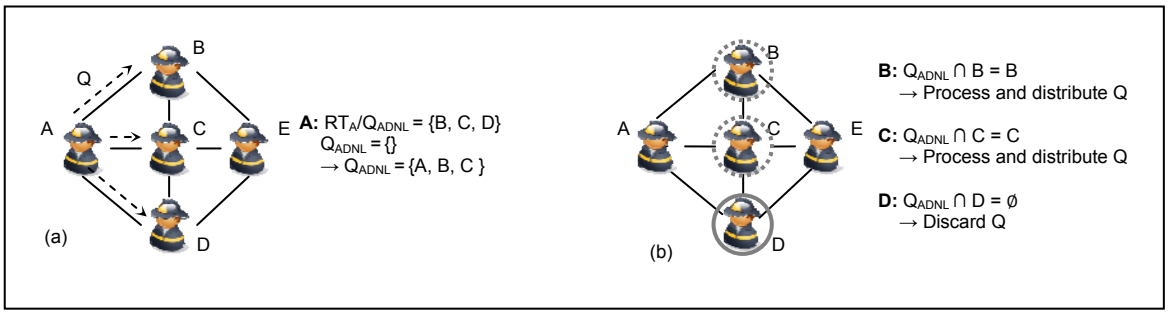

Figure 6.4. RECOWER Query Distribution Suppression.

6.6.4, we delve into details to present the main self-adaptive mechanisms introduced by RECOWER, namely *QoC-based context data caching* and *adaptive selection of broadcast neighbours*.

6.6.1. RECOWER Software Architecture

Figure 6.5 presents the local software architecture adopted by the RECOWER CDDI. Following the logical model presented in Section 4.1, we organize it in two main layers: a *Context Data Management Layer* and a *Context Data Delivery Layer*.

The Context Data Management Layer implements all the high-level functionalities related to context data production and access. Every context aspect is mapped to a particular context type that describes the layout of its own data: since RECOWER exploits an object oriented context model [55], the type definition describes the fields involved in each data instance. All current context types are stored and available by means of a local *Context Data Type Storage*. Finally, each context data type is associated with a proper *Context Data Module* that contains 1) a *Source* to realize data injection; 2) a *Sink* to enable data retrieval; and 3) a *Context Data Cache* to store locally cached context data.

The Context Data Delivery Layer implements all the low-level functionalities related to context data routing and wireless communications. The *Communication Module* offers technology-independent send/receive operations while proper adapters, i.e., the *WiFi* and the *BT Adapter*, map them to the real low-level technology-dependent ones. Each adapter implements a *Neighbourhood Sampling* module that supplies access to the current available one-hop neighbours. Finally, the *Adaptive Routing Manager* realizes context data routing. It receives context data requests both from local and remote context modules, and passes context queries to the proper module that, in its turn, handles the matching phase with locally stored data.

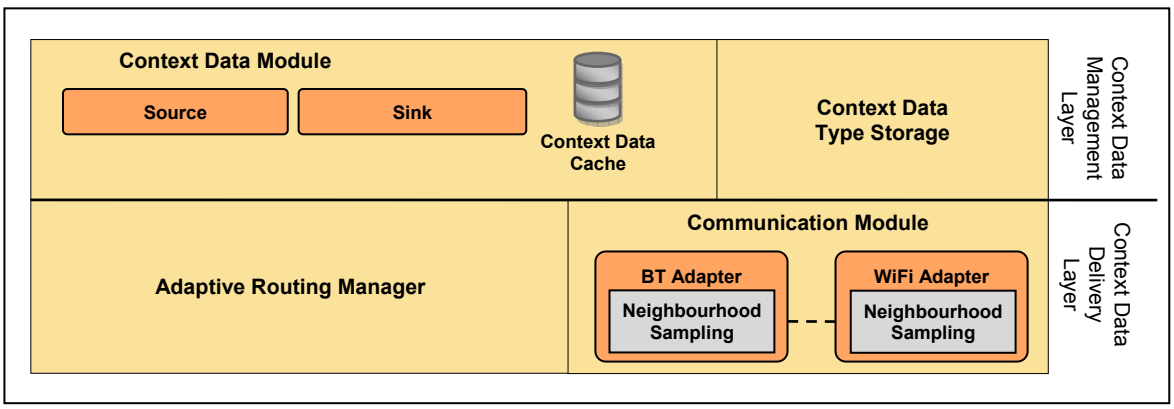

Figure 6.5. RECOWER Software Architecture.

6.6.2. QoC-based Context Data Caching

As presented in Section 6.5.1, RECOWER context data caching is adapted at runtime depending on data quality attributes and QoC constraints of close neighbours [106]. With this goal in mind, our CDDI exploits a three phase algorithm where each mobile node 1) monitors close neighbours to know their own quality classes; 2) merges collected quality classes to obtain a final quality class, namely a *cache quality class*; and 3) reconfigures all local caches to keep only data respecting the cache quality class constraints. In this section, we better present the main phases involved during cache reconfigurations at runtime.

In the first phase, each mobile node collects the user quality classes associated with close neighbours. The neighbourhood considered during cache reconfigurations could also span multiple hops, but there are two strong contraindications against this solution. First, since query TTL assumes different values according to the desired service retrieval scope, an upper bound to the neighbourhood that can route queries to a particular node is impossible to find. Second, due to high node mobility and density, a consistent view over multi-hop neighbourhood is difficult to reach with a low overhead. Hence, we decided to limit the influencing neighbourhood to one-hop nodes, in order to better trade off management overhead with performance gain. Finally, to enable the collection of the quality classes associated with neighbours, RECOWER piggybacks them in the mobility beacons periodically emitted by each mobile device.

In the second phase, RECOWER merges quality classes collected by neighbours to obtain the final cache quality class that, in its turn, will determine the quality constraints used to anticipate the removal of data with low quality. By exploiting the hierarchical organization of the different quality classes explained in Section 6.5.1, different merging policies, such as MIN and MAX, can be adopted. However, MIN can easily impose the lowest quality class, thus leading to frequent replacements and data with poor quality. Instead, MAX can easily impose the highest quality class, thus leading to empty caches and suboptimal resource usage. Hence, RECOWER adopts the subsequent approach. If QC_{SENDER} is the class associated with the node that is currently reconfiguring its own local caches, RECOWER examines all the quality levels from the highest one to QC_{SENDER}. For each quality class, a local accumulator is incremented with the number of neighbours that belong to that class. When the accumulator becomes higher than a threshold λ, obtained by scaling neighbourhood size, the associated class is used as final cache quality class. Let us

note that the final cache quality class could also be equal to QC_{SENDER} if the accumulator never reaches the threshold λ. With this approach, RECOWER will never choose a cache quality class lower than QC_{SENDER}, hence, a node will never cache data that do not respect its own quality level.

Finally, in the third and last phase, each node reconfigures its local cache with the selected cache quality class. Associated quality constraints are used in a two-fold manner. First, they define the cache admission policy: hence, from now on, if the cache is full, data that do not respect them will never be cached. Second, they choose the element to evict when necessary. If we have to remove an element, we first select all the elements that do not respect current cache quality class, and then we choose the element to evict by a traditional LRU policy; if all the data respect current cache quality class, we then select the element with the lowest quality, by also scaling and weighting quality attributes in different ways, according to their relative significance. During reconfigurations, until there is free space, we temporary keep previous context data even if out-of-QoC, in order to further increase context data availability.

To conclude, every time a mobile node finds itself in an area populated by nodes having higher quality classes, the adopted merge operation forces it to cache context data with higher quality. Hence, our QoC-based caching algorithm effectively tailors context caching at runtime, so to improve the overall quality of the data in the physical area.

6.6.3. Adaptive Selection of Broadcast Neighbours

The adaptive query flooding is based on a selection phase useful to identify the neighbours that will receive the query [107]. In the current implementation, neighbour selection is driven by query storage load factors (i.e., the memory available on neighbours) and data repositories diversity (i.e., the parameter that measures the diversity between the local data repositories and the ones deployed on one-hop neighbours). To avoid additional messages, the management information needed by the adaptive distribution process is piggybacked into mobility beacons. When a node sends its own beacon message, it piggybacks three parameters (see Figure 6.6 for the associated pseudo code): 1) a *Local Query Load Factor (LQLF)*; 2) a *Data Key List (DKL)*; and 3) a *Data Repositories Diversity Factor (DRDF)*. The LQLF is the ratio between the number of locally stored queries and Q_{MAX}: hence, it is in the range [0; 1], and higher values indicate overloaded situations. The DKL is the list of the keys of locally memorized data, and it is used to evaluate diversity with close context data repositories. Finally, the DRDF is the average

Variables

- localNodeID: logical id associated with the current node
- N: the current set of physical neighbors
- Q: the current set of stored queries
- R: repository of local context data
- R[i]: i^{th} data in the local repository; $i \in [0; D_{MAX})$
- MgmtInformation[n]: map of the management information <LQLF, DKL, DRDF> for node n

Function

- **storeQuery(Query q):** memorizes q into the local support and schedules further distributions if required
- **piggybackOnMobilityBeacon(Message m):** piggybacks message m in the next mobility beacon sent to all one-hop neighbors
- **scheduleSendData(Data d, NodeID n):** send data d to node n in a random delay less than DRD
- **lookupLocalQueryCopy(Query q):** checks if q is already known. If yes, returns the local copy of the query
- **broadcastQuery(Query q):** broadcast q to the current one-hop neighborhood

Messages

- **STATUS<LQLF, DKL, DRDF>**: message containing the management information required by the adaptive data distribution solution

```
Invoked every beacon period
void sendMgmtInformation ()
1: Build m = STATUS<|Q|/Q_MAX, buildLocalDKL(),
       calculateDRDF()>;
2: piggybackOnMobilityBeacon(m);

float calculateDRDF()
1: float localDRDF = 0.0;
2: List<DataKey> localDKL = buildLocalDKL();
3: for all n ∈ N; do
4:    localDRDF+= (1 − |localDKL ∩ MgmtInformation[n].DKL| / |localDKL ∪ MgmtInformation[n].DKL|);
5: return localDRDF/ |N|;

List<DataKey> buildLocalDKL()
1: List<DataKey> l;
2: for all d ∈ R; do
3:    l.add(d.key);
4: return l;

List<NodeID> calculateUnreachedNeighbors(Query q)
1: List<NodeID> unreachedNeighbors = {};
2: for all n ∈ N; do
3:   if (n ∉ q_ADNL); then
4:    unreachedNeighbors = unreachedNeighbors ∪ n;
5: return unreachedNeighbors;

Distribute query q
void distributeQuery(Query q)
1: List<NodeID> feasibleNeighbors =
       calculateUnreachedNeighbors(q);
2: if (feasibleNeighbors.isEmpty()); then
3:    return;
4: List<NodeID> logicalNeighbors =
       selectLogicalNeighbors(feasibleNeighbors);
5: if (logicalNeighbors.isEmpty()); then
6:    return;
7: q_ADNL = q_ADNL ∪ logicalNeighbors;
8: broadcastQuery(q);

Received query q from node n
void receiveQuery(NodeID n, Query q)
1: for all d ∈ R; do
2:    if (q.match(d)); then
3:       scheduleSendData(d, n);
4: Query lqc = lookupLocalQueryCopy(q);
5: if (lqc != NULL); then
6:    lqc_ADNL = lqc_ADNL ∪ q_ADNL;
7:    return;
8: if (!q_ADNL.contains(localNodeID)); then
9:    return;
10: if (!q.isSatisfied ()); then
11:    storeQuery(q);

List<NodeID> selectLogicalNeighbors(List<NodeID>
     feasibleNeighbors)
1: float averageLQLF = 0.0;
2: for all n ∈ feasibleNeighbors; do
3:    averageLQLF+= MgmtInformation[n].LQLF;
4: averageLQLF /= |feasibleNeighbors|;
5: int logicalNeighborhoodCardinality;
6: if (averageLQLF < γ); then
7:    logicalNeighborhoodCardinality = |feasibleNeighbors|;
8: else
9:    logicalNeighborhoodCardinality =
          ((1−|feasibleNeighbors|)/(1−γ)) × averageLQLF + ((|feasibleNeighbors|−γ)/(1−γ));
10: if (logicalNeighborhoodCardinality != |feasibleNeighbors|); then
11:    List<NodeID> limitedFeasibleNeighbors;
12:    Order feasibleNeighbors according to associated DRDFs;
13:    limitedFeasibleNeighbors.copyHighestElements(
           feasibleNeighbors, logicalNeighborhoodCardinality);
14:    return limitedFeasibleNeighbors;
15: else
16:    return feasibleNeighbors;

Received msg STATUS<LQLF, DKL, DRDF> from node n
void receivedMgmtInformation (n, LQLF, DKL, DRDF)
1: MgmtInformation[n] = <LQLF, DKL, DRDF>;
```

Figure 6.6. Adaptive Query Flooding Pseudo-code.

diversity between the data repositories at the sender node and the repositories available in its own one-hop neighbours (see Figure 6.6); its value is in [0; 1], and higher values are better since associated with higher data repositories diversity.

When a node has to broadcast a query, first it selects the cardinality of the set of neighbours that will receive it. As showed in the function *selectLogicalNeighbors* in Figure 6.6, it calculates an *averageLQLF* as the average of the LQLFs collected by the nodes that had not already received the query (the node that had already received the query are not considered since they will be not affected by the current distribution). If this value is lower than a particular threshold γ, all the current neighbours will receive and process the query. Otherwise, RECOWER finds the cardinality of the final neighbours set through a linear function (as showed in Figure 6.6), and selects involved nodes by exploiting collected DRDFs. As limited research scopes increase the probability of missing important data, RECOWER sends the query to the neighbours with the highest DRDF values, so to hit the ones that, by having high data repository diversity with their own neighbours, can

reach a wider set of context data.

6.6.4. Optimized Management Data Representation

We optimize the representation of all those management data useful to implement aforementioned self-adaptive mechanisms, so to reduce the runtime management overhead. Toward this goal, we exploited Bloom filters [108, 109]; for the sake of completeness, we now briefly introduce the main properties of this data structure.

A Bloom filter is a space-efficient probabilistic data structure that supports membership queries on a set $A=\{a_1, a_2, \ldots, a_n\}$ of n keys. Each filter consists of a vector of m bits, initially all set to 0. Each key of the original set passes through k independent hash functions $\{h_1,h_2,\ldots,h_k\}$ with output in [0; m-1]. The filter associated with the keyset is obtained by setting to 1 all bits at positions $h_1(a)$, $h_2(a)$, ..., $h_k(a)$ for each element $a \in A$. Given a generic key b, we check all the bits in $h_1(b)$, $h_2(b)$, ..., $h_k(b)$ and, if any of them is 0, then certainly b is not in the original set. Otherwise, we assume that b is in the set, although it may not be the case because Bloom filters may present false positives. However, if we assume that adopted hash functions have a uniform distribution, the false positive ratio is roughly equal to $0.6185^{m/n}$: thus, given an upper bound to |A|, we can reduce false positives by increasing filter length.

Due to the aforementioned good properties, RECOWER uses the probabilistic membership test of Bloom filters to optimize the representation of both query ADNL and DKL. First, since query ADNL is modified by only inserting elements, and it is used to only perform membership tests, it can be easily implemented with a Bloom filter. In addition, the append operation required when a node receives an already known query is equal to a simple bit-wise OR between the already known ADNL and the ADNL carried by the received query. This is one of the most appealing properties of a Bloom filter: the filter associated with the union of two different sets is equal to the bitwise OR of two different filters, each one associated with each set [109]. Second, the DKL is mainly used to evaluate repositories diversity. Unfortunately, since an inverse mapping from a Bloom filter to original keys is impossible, we need to estimate DRDF in a probabilistic manner. Even if the literature offers probabilistic bounds to address the problem of intersection estimation between two Bloom filters, they depend on employed hash functions and on properties of the original data keyset. To reduce the computational load, we adopted a coarse-grained solution in which the diversity between two repositories is approximated by the diversity of the two associated Bloom filters. Consequently, RECOWER calculates

the diversity between two different repositories by using the formula (6.2) instead of the formula (6.1) (see Figure 6.6).

$$\left(1 - \frac{|\text{localDKL} \cap \text{MgmtInformation[n].DKL}|}{|\text{localDKL} \cup \text{MgmtInformation[n].DKL}|}\right) \tag{6.1}$$

$$\left(\frac{\sum_{i=0}^{i=m-1} (\text{localDKL[i]} + \text{MgmtInformation[n].DKL[i]})\%2}{m}\right) \tag{6.2}$$

In other words, it compares bit-by-bit the two Bloom filters by considering a positive unitary increment when the compared bits differ. Finally, the obtained value is divided by the filter length to normalize it. If the filters are completely different, i.e., they do not share two equal bits at the same position, the final diversity is 1; otherwise, if two filters are exactly equal, the diversity is 0. Of course, this estimation is suboptimal since there is no direct one-to-one relation between two equal bits into the Bloom filters and the number of elements into the intersection.

6.7. Simulation-based Results

To assess the technical soundness of our proposals, we implemented RECOWER and all the aforementioned mechanisms in the network simulator NS-2.34. We considered an area of 350x350m with 50 nodes, wireless ad-hoc links based on IEEE 802.11g technology (bandwidth of 54 Mbps) with a transmission range of 100m, and Two Ray Ground as propagation model. Also, each node emits a mobility beacon every 10 seconds to signal its presence to one-hop neighbours.

As regards mobility modeling, few works in literature proposed complex solutions for disaster area scenarios [97, 110]. However, they dealt with the whole disaster area: if we focus on the incident area, to the best of our knowledge all the research works in literature model node movement with a Random WayPoint (RWP) model. Hence, since RECOWER concerns context-aware services into the incident area, we adopted RWP with the following parameters: uniform speed in [1; 2] meters/second (pedestrian velocity) and a uniform distributed pause in [0; 10] seconds before selecting the next waypoint. Each node selects the next waypoint before reaching area borders (no node departures and arrivals); this border rule resembles real scenarios where the same fireman carries an injured person out of the incident area, and then comes back to find other humans. Finally, simulations last 900 seconds, and all reported results are average values over 33 test executions to obtain a good confidence; standard deviation is also showed to evaluate results dispersion.

In the remainder, we present the experimental results obtained in such settings. For

the sake of clarity, we divided them in two main subsections: Section 6.7.1 is focused on local context data management, while Section 6.7.2 presents results concerning adaptive query distribution.

6.7.1. Quality-based Context Data Caching Evaluation

To test the quality-based context data caching approach, we need to model both context data and query production. As regards context data production, we fairly divided a set of 1000 context data sources between all the mobile nodes, hence, each node produces 20 context data. If not stated differently, in the following we use a D_{MAX} parameter of 30: 20 elements are reserved to store the last version of locally produced data, while the other 10 elements are occupied by data received by neighbours according to locally issued queries. Each context data instance has an application payload of 3KB, so as to simulate challenging scenarios where the context data may contain compressed images about the incident area or complex context data. In addition, each context data source periodically produces a new context data instance; if not stated otherwise, each instance has a FL parameter of 300 seconds, thus representing quite stable context aspects. Finally, as stated before, each data instance has an up-to-dateness quality attribute equal to the ratio between RL and FL parameters (hence, it is in [0; 1] and values closer to 1 are better).

As regards query production, we divided mobile nodes in 2 different quality classes: the first one, QC_1, contains 25 nodes, and accepts only data with up-to-dateness higher than 0.7; the second one, QC_2, contains the remaining 25 nodes, and accepts all possible up-to-dateness values. Each mobile node emits a fixed number of queries for each second, by uniformly selecting one context data source over the 1000 available. This represents the worst case scenario since context data caching usefulness is largely reduced; at the same time, we believe that this models a large set of realistic workloads in such scenarios (for instance, access to the localization information of a single first responder, retrieval of health information associated with a single person, etc.). All the queries are flooded without any of the optimizations previously presented and with a data retrieval time of 2 seconds. Finally, α and β parameters, used to calculate the final random routing delay applied at each mobile node, are respectively 0.7 and 0.9.

Before discussing the results, we want to remark that we are evaluating a worst case scenario since: 1) each node emits requests with a uniform distribution, thus reducing the probability of finding data on near neighbours; 2) context data are stored in a decentralized MANET, and partitions can significantly reduce context data availability; and 3) since

context data are distributed only as consequence of matching queries, many queries have to reach the data creator node before obtaining a positive response.

In the first set of experiments, we compare our QoC-based caching algorithm with a simple LRU under uniform access patterns. In the remainder, the threshold λ used to find the final cache quality class is equal to the number of neighbours divided by 3. By using a request rate of 0.5 reqs/s and a query TTL in {1, 2, 3}, Figure 6.7 (a), Figure 6.7 (b), and Figure 6.7 (c) respectively represent the average retrieval time, the percentage of satisfied queries, and the average up-to-dateness of retrieved data; for the sake of clarity, results are divided according to the different quality classes, since associated quality constraints greatly affect final experienced performance. To draw some conclusions, although the two approaches lead to very similar average retrieval times (see Figure 6.7 (a)), the quality-based approach always ensures higher percentages of satisfied queries than simple LRU (see Figure 6.7 (b)). In fact, our quality-based approach tends to keep higher quality data, i.e., data matching both QC_1 and QC_2 constraints, thus finally leading to a higher number of satisfied queries for both classes. It is worth noting that this increased reliability is negligible when query TTL is 3, as that value is associated with a network-wide distribution scope: in fact, if each hop covers 100 meters, a query distributed with a TTL of 3 can potentially reach any point in the network, thus finding the mobile node that hosts the wanted context data source. Finally, focusing on Figure 6.7 (c), let us remark that, of course, QC_1 clients find context data always with up-to-dateness higher than 0.7 due to associated constraints. At the same time, our quality-based approach improves average up-to-dateness of the data found by QC_2 clients. Surprisingly, it slightly reduces up-to-dateness of the data found by QC_1 clients, but this is mainly due to the increased number of satisfied queries from context data cached on close peers, that usually have reduced quality attributes.

In the second set of experiments, we modify D_{MAX} to test how this parameter affects algorithm performance. By using above parameters and a TTL of 2, Figure 6.8 (a), Figure

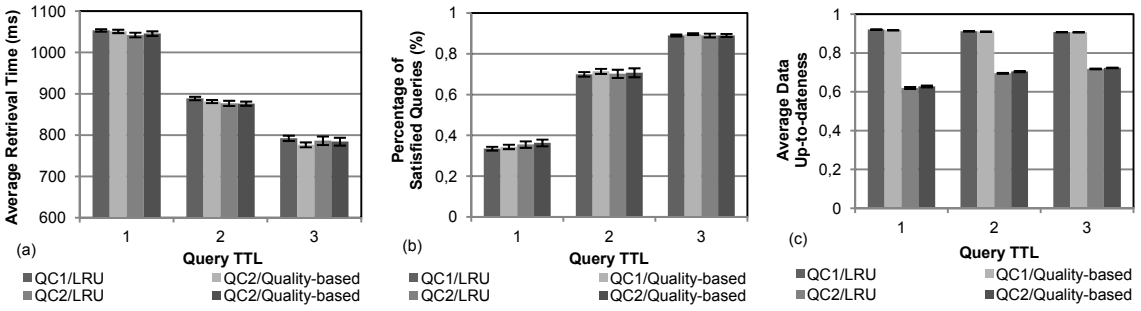

Figure 6.7. LRU vs. Quality-based Caching with Uniform Access Patterns and Different Query TTL.

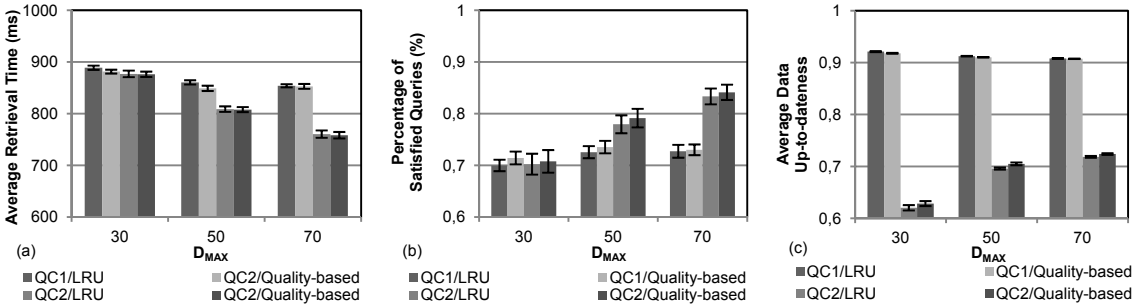

Figure 6.8. LRU vs. Quality-based Caching with Uniform Access Patterns and Different D_{MAX}.

6.8 (b), and Figure 6.8 (c) show respectively the average retrieval time, the percentage of satisfied queries, and the average up-to-dateness for each quality class when D_{MAX} is equal to {30, 50, 70}. From Figure 6.8 (a), we remark that, of course, higher D_{MAX} values lead to reduced average retrieval times: in fact, bigger repositories lead to more copies of the same context data instance into the network, thus increasing the probability of finding matching data closer to the query sender node. Also, from Figure 6.8 (b), we note that higher D_{MAX} values increase the percentage of satisfied queries, since each node can reach a wider set of data cached in the physical proximity. It is worth noting that aforementioned variations are more visible for QC_2 clients since they also accept context data with very low quality; instead, QC_1 clients require high quality data, hence, they are less sensible to D_{MAX} values since not all the cached data will match their own quality constraints. Finally, in line with the results of Figure 6.7 (c), Figure 6.8 (c) shows that our quality-based approach leads to matching data with higher up-to-dateness values.

In the last set of experiments, we evaluate the two caching algorithms with different data FL parameters, so to better assess their performance with more dynamic context data. Figure 6.9 (a), Figure 6.9 (b), and Figure 6.9 (c) show the same set of results used in previous experiments, with a data FL parameter in {900, 450, 300, 225, 180} seconds. As we noted in previous experiments, the quality-based approach performs better than simple LRU. Figure 6.9 (a) shows that short lived data tend to increase average data retrieval times; in fact, in that case, mobile nodes proactively delete context data due to RL

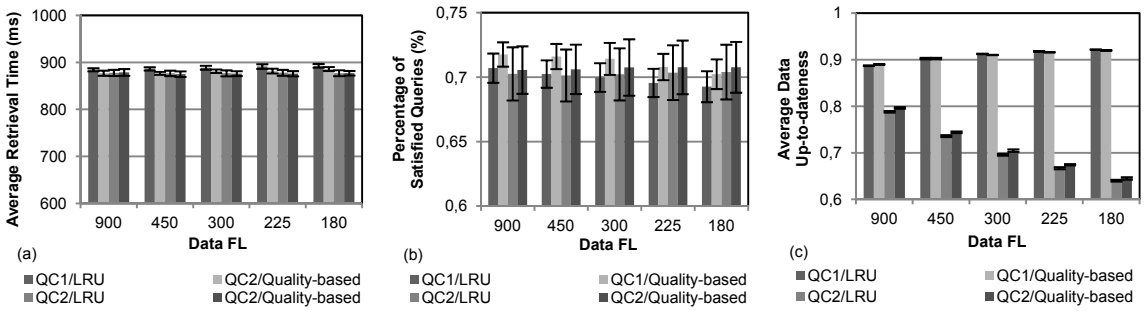

Figure 6.9. LRU vs. Quality-based Caching with Uniform Access Patterns and Different Data FL.

expiration (see Section 6.3), and end up with not exploiting all the maximum cache size. At the same time, from Figure 6.9 (b), we note that the percentage of satisfied queries reduces with short lived data. This is more visible for QC_1 clients due to tighter quality constraints that, in their turns, lead to more frequent accesses to real context sources. Finally, Figure 6.9 (c) shows that short lived data lead to lower up-to-dateness values, especially for QC_2 clients. In fact, short lived data have fast decreasing up-to-dateness values, hence, queries that match context data cached on close mobile nodes have higher chances of finding data with reduced quality.

In conclusion, from above results we conclude that the quality-based approach usually performs better than simple LRU in terms of percentage of satisfied queries and average up-to-dateness. In addition, it only requires the dissemination of the quality class of each mobile node; that is easily accomplished by piggybacking node quality class in its own mobility beacon. Hence, considering the very low network overhead introduced by our approach, we think that it is a feasible and viable choice to increase the quality of the context data cached in a physical area, so as to better exploit precious storage resources.

6.7.2. Adaptive Query Flooding Evaluation

RECOWER context query distribution is fundamental to build context data distribution paths, and can greatly affect context data availability. In this section, we test our adaptive query flooding protocol [107], and we compare it with a traditional flooding approach. NS2 simulations have the same parameters adopted in previous section. In particular, the WiFi channel exploits IEEE 802.11g parameters with a total available bandwidth of 54 Mbps. Each mobile node has a D_{MAX} of 30 and hosts 20 different context data sources, that produce a single context data instance at the beginning of the simulation with an FL parameter of 900 seconds and a payload of 3KB. As for quality-based distribution, we use a data retrieval time of 2 seconds and no constraints on data up-to-dateness, and each mobile node periodically emits a new query by using a uniform distribution to select the source. Finally, if not stated differently, simulations use α and β parameters respectively equal to 0.7 and 0.9, and the threshold γ, used by our adaptive query flooding to reduce query replication, is set to 0.5.

In the remainder, we compare our Adaptive Flooding (AF) algorithm with Naïve Flooding (NF) under different request rates and TTL values. In NF, each node simply broadcasts a query depending on the associated TTL, while always introducing proper random query/data routing delays. Of course, in both NF and AF, if the received query is

already known, the node does not broadcast it again to avoid the introduction of additional traffic; although infinite loops are not possible due to the limited TTL, if a node broadcasts again the same query, it will probably hit the same set of neighbours, thus introducing useless overhead.

In the first set of experiments, we focus on an ideal situation where Q_{MAX} is infinite, namely each mobile node stores all the received queries with no replacement. As the ratio between the number of stored queries and Q_{MAX} is used by AF to reduce query replication, this test condition setups a worst-case scenario for AF since it will never explicitly reduce query distribution scope. Hence, differently from NF, AF will only prevent the distribution of a query when all the current physical neighbours had already received the query. Figure 6.10 (a), Figure 6.10 (b), and Figure 6.10 (c) show respectively average retrieval times, percentage of satisfied requests, and percentage of dropped packets with request rate in {0.5, 1, 2} reqs/s and different flooding algorithms. To draw important conclusions, if the network load is low (for instance, when the TTL is 1), NF and AF performs very similarly. Instead, when the network load increases due to both higher TTL values and higher request rates, AF always performs better than NF: in fact, it ensures lower retrieval times, higher percentages of satisfied queries, and lower dropped packets. All these positive effects are mainly connected with the reduced number of distributed queries that, in its turn, reduces the probability of message collision and network congestion. Finally, with a request rate of 2 reqs/s, the usage of a query TTL equal to 3 surprisingly reduces satisfied queries; that is clearly in contrast with the bigger query distribution scopes ensured by such TTL parameter. Considering that we do not have memory limitations, this effect is due to the increased number of dropped packets: in fact, several queries are dropped due to message collisions (we recall that broadcast messages are more collision-prone than unicast ones as they do not exploit the RTS/CTS mechanism), thus preventing the building of distribution paths into the MANET.

In the second set of experiments, we consider more realistic scenarios where the

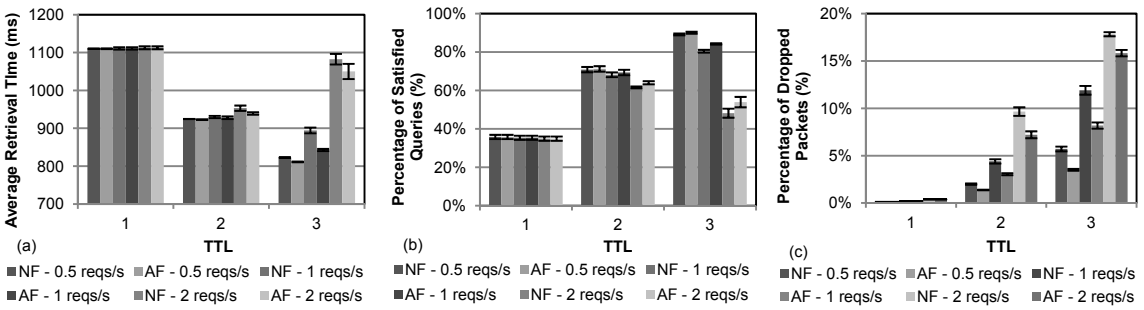

Figure 6.10. Comparison between Naïve and Adaptive Flooding.

number of queries that can be stored on each mobile device is limited by a Q_{MAX} parameter equal to 50. By exploiting the same parameters of previous experiments, Figure 6.11 (a), Figure 6.11 (b), and Figure 6.11 (c) represent the same set of results for the new scenario with query storage limitation. Of course, similarly to Figure 6.10, when the final network load is limited due to low TTL and/or request rate values, NF and AF perform very similarly. Instead, when the network load increases, AF performs better than NF in all the considered test scenarios. As each node has a Q_{MAX} parameter of 50, differences between NF and AF start to become visible when the expected number of stored queries at each mobile node is higher than 25, i.e., when the averageLQLF is higher than the γ parameter of these experiments. Although Figure 6.11 (a) and Figure 6.11 (b) suggest that NF and AF significantly differ only with TTL equal to 3 and request rates in {1, 2} reqs/s, Figure 6.11 (c) shows that AF always ensures reduced dropped packets starting with TTL equal to 2. This does not have a direct impact on the percentage of satisfied queries due to high path replication.

From above results, we conclude that AF always outperforms NF in all situations (also when there are no memory limitations). Hence, in the remainder, we focus on AF evaluation only, to better study the influence of data retrieval time and Q_{MAX} parameter over the performance indicators considered before.

In the third set of experiments, we evaluate the effect of the QoC data retrieval time on AF query distribution protocol. In fact, higher data retrieval times result in higher maximum routing delays applied at each mobile node; that, in its turn, may increase the randomness of applied routing delays, thus favouring query distribution suppression. Figure 6.12 (a), Figure 6.12 (b), and Figure 6.12 (c) represent final performance results with data retrieval time in {2000, 3000, 4000} ms, obtained by using a request rate of 2 reqs/s and a Q_{MAX} of 50. Let us remark that we considered such parameters since they lead to realistic scenarios, where the number of queries to be stored at each mobile node can be higher than Q_{MAX}. Starting from Figure 6.12 (a), of course, higher QoC data retrieval

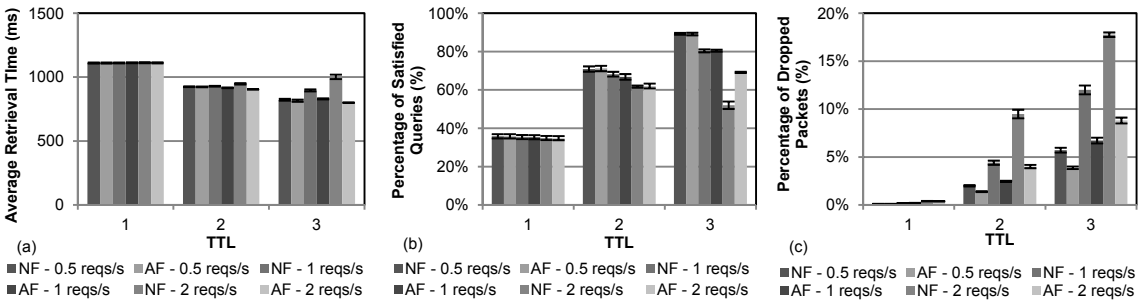

Figure 6.11. Comparison between Naïve and Adaptive Flooding with Memory Limitations.

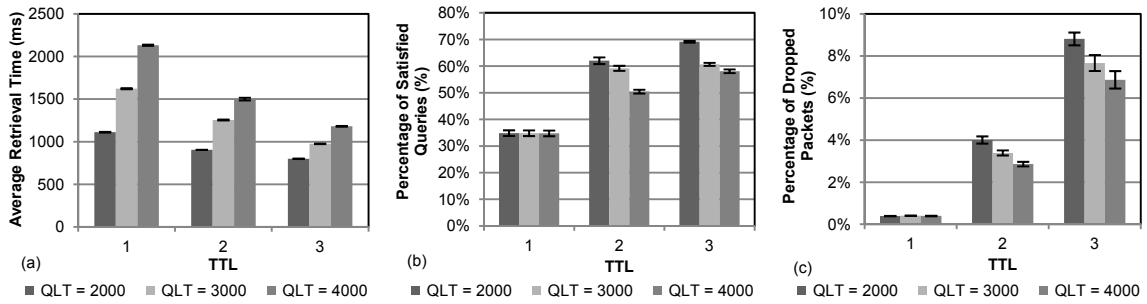

Figure 6.12. Effects of QoC Data Retrieval Time on Adaptive Flooding.

times lead to higher average retrieval times due to bigger routing delays applied at each mobile node. At the same time, from Figure 6.12 (b), we remark that, apart from the case of TTL equal to 1, higher retrieval times reduce the percentage of satisfied queries. In fact, higher data retrieval times increase the number of queries potentially stored on each mobile node, and that increases averageLQLF (as presented in Figure 6.6) and reduces context query replication into the MANET. From Figure 6.12 (c), we note that higher QoC data retrieval times lead to reduced packet droppings, but that is mainly due to reduced network traffic consequence of query storage limitations. Hence, we remark that, before a real production phase, it is important to correctly estimate the number of queries emitted by each mobile node, as well as node density and average query lifetime, to correctly choose the Q_{MAX} parameter; a wrong estimation of this value can significantly reduce the reliability of the context data distribution process.

Finally, in the last set of experiments, we investigate the effects of the Q_{MAX} parameter on context query distribution reliability. By using a request rate of 2 reqs/s and a data retrieval time of 4000 ms, Figure 6.13 (a), Figure 6.13 (b), and Figure 6.13 (c) respectively show the average retrieval times, the percentage of failed requests, and the percentage of dropped packets for TTL in {1, 2, 3} and Q_{MAX} value in {50, 100, 150, No limit}. Focusing on Figure 6.13 (b), we remark that, if TTL is 2, higher Q_{MAX} values result in higher reliability. Instead, when TTL is 3, the scenario with no Q_{MAX} limitation results in the worst reliability; in fact, as also confirmed by Figure 6.13 (c), that scenario is

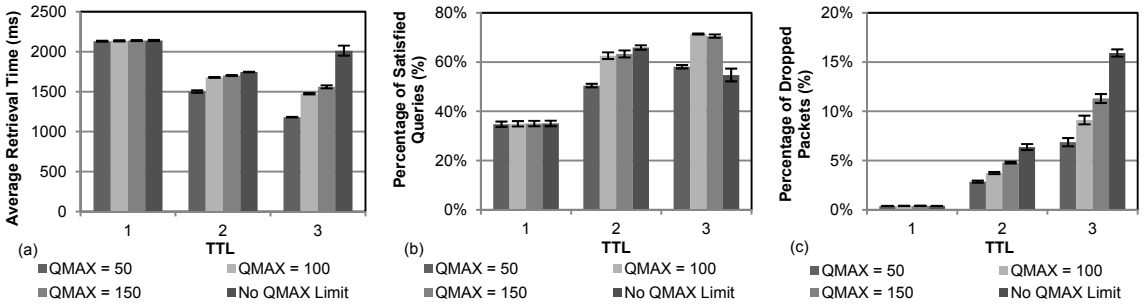

Figure 6.13. Effects of Q_{MAX} on Adaptive Flooding.

associated with the higher percentage of dropped packets. Hence, while reasonable Q_{MAX} values can increase distribution process reliability, excessive query replication can mine it due to the increased network congestion; in fact, from Figure 6.13 (c), we note that the percentage of dropped packets is proportional to the Q_{MAX} value.

To conclude, the adaptive flooding solution always outperforms the naïve one. The reduced number of distributed queries saves network bandwidth, and positively affects both the scalability and the reliability of the context data distribution. Even if this solution can lead to reduced research scopes due to avoided query distributions, both the small number of collisions and the usage of routing paths with high data repositories diversity make our solution valid for supporting context data distribution.

7. Context Data Distribution in Smart University Campus Scenarios

After the presentation of the RECOWER CDDI, mainly based on ad-hoc communications, this chapter considers the realization of context-aware services in hybrid network deployments, in which we have both infrastructure-based and ad-hoc wireless communications. In particular, here we focus on the design, the implementation, and the deployment of context-aware services for smart university campuses. In Section 7.1, we open this chapter by better stating both the main issues and the design guidelines considered by our CDDI, named SALES, for smart university campuses. Then, in Section 7.2, we present the hierarchical distributed architecture, while Section 7.3, Section 7.4, and Section 7.5 detail the main solutions adopted at the different CDDI logical layers. Finally, Section 7.6 thoroughly presents the implementation details of our SALES real-world prototype, and Section 7.7 concludes the chapter by reporting extensive experimental results useful to better assess the technical soundness of our proposals in real-world deployments.

7.1. SALES CDDI

Context-aware services for smart university campuses are receiving a lot of attention in the last years. At the current stage, there are already different universities offering context-aware services to their own students: to mention few, both ActiveCampus and SmartCampus exploit context-aware capabilities to offer innovative services into the university campus [9, 10]. Such services can greatly enhance the social experience into the campus, for instance, by suggesting possible friendships and study groups. Although in these scenarios the CDDI can rely on fixed wireless infrastructures for the sake of context provisioning, challenging issues have to be addressed both at the context data delivery and at the context data management layer.

Starting with the delivery layer, such systems can introduce high network traffic to deliver context data. Let us note that, differently from RECOWER, reliability is not a first concern as campus services do not present risks for human lives; of course, both service interruptions and packet droppings into the context distribution process can degrade the final user experience, but they can be tolerated and controlled to reach better tradeoffs between scalability and quality. Instead, similarly to what happened in RECOWER scenarios, here we can experience very high node densities: a university classroom, where

several students use their own devices to run context-aware services, is a clear and frequent deployment with high node density that we can find in smart university campuses. Extremely crowded areas are challenging from the scalability viewpoint, since wireless bandwidth is shared by all mobile nodes and consequent wireless collisions and congestions can easily degrade final service quality. In addition, fixed wireless infrastructures should be carefully used since they are usually exploited to provision other kinds of services, such as Internet connectivity, video streaming services, and so forth, to roaming users, and should be not overloaded by context data distribution. Moving to the context data management layer, these systems have to handle large amounts of context data: differently from disaster area scenarios, where context data are used to quickly detect dangerous situations, here context-aware services can require heavy context processing operators. Even if the storage of such data can be considered not a first concern due to the availability of powerful fixed servers, both context data aggregation and filtering operators can introduce high management overhead.

Hence, to support context provisioning in such scenarios, the CDDI should exploit some fundamental guidelines. Although some of them are similar to the ones we already discussed in RECOWER (see Section 6.1), here the availability of both a fixed infrastructure and physical servers possibly changes how we actually apply them.

First, considering the delivery layer, we mainly confirm the design guidelines presented in RECOWER. Such a CDDI should exploit heterogeneous wireless technology for the sake of scalability and system coverage. In addition, it should integrate heterogeneous wireless modes, i.e., infrastructure-based and ad-hoc communications, so to effectively reduce the management overhead on fixed infrastructures by favouring cooperative context distribution among close neighbours.

Second, as regards the context data management layer, the CDDI should exploit a distributed data repository based on both mobile and fixed nodes. Context data should be cached and replicated into the distributed architecture with the main goal of reducing access times and network overhead. Even if the storage of all the context data is probably feasible by only exploiting fixed servers, the CDDI should memorize context data on mobile nodes, so that they can share them with close neighbours by ad-hoc links.

Third, the CDDI should exploit physical and logical locality principles to tailor context data memorization and distribution. Differently from disaster area scenarios, in this case the CDDI can assume the availability of anchor nodes, e.g., WiFi APs deployed into the university campus, that can supply relative localization information useful to

enforce the physical locality principle. Also, the availability of full-fledged physical servers enables the realization of complex algorithms that, by monitoring context data requests, can identify logical localities between mobile nodes and, accordingly, reconfigure the distribution process at runtime.

Finally, the CDDI should adapt to available resources and QoC constraints: mobile devices have limited resources in terms of CPU, memory, and network connectivity; hence, a CDDI should respect the minimum intrusion principle by guaranteeing the introduction of limited and controlled overhead [63]. We remark that, since offered context-aware services are not safe-critical, mobile users would probably tolerate inconsistencies in context view and slightly quality degradations, but not fast battery depletion and heavy computational load. In addition, similarly to what we discussed in RECOWER, also here the CDDI should introduce and enforce different quality constraints, for instance, to support different user roles (e.g., students and professors); at the same time, it should self-adapt and manage its internal behaviour according to available resources. For instance, a CDDI should automatically and dynamically reconfigure its internal facilities, such as differentiated context data routing and decentralized caching, by also finely tuning memory and computing resources (number of processes, communication data rate, …) depending on current working conditions (available bandwidth, number of context exchanges, …), number and class of clients, and agreed quality contracts [111].

Following above guidelines, we designed our CDDI for smart university campuses, called Scalable context-Aware middleware for mobiLe Environments (SALES) [112]. In the remainder, by following a presentation order similar to the one adopted in Chapter 6, we introduce additional details and design choices associated with SALES at the different layers.

7.2. A Proposed Distributed Architecture

To ease the enforcement of the physical locality principle, we decided to adopt the three-level tree-like distributed hierarchical architecture showed in Figure 7.1 (a grey line between two nodes means that they can exchange messages directly). SALES CDDI spans all the involved devices, both fixed and mobile, and exploits mobile devices as temporary data carriers to perform context data routing and caching.

For the sake of infrastructure management, we distinguish four main types of nodes with different responsibilities. In particular, by considering the different hierarchical levels

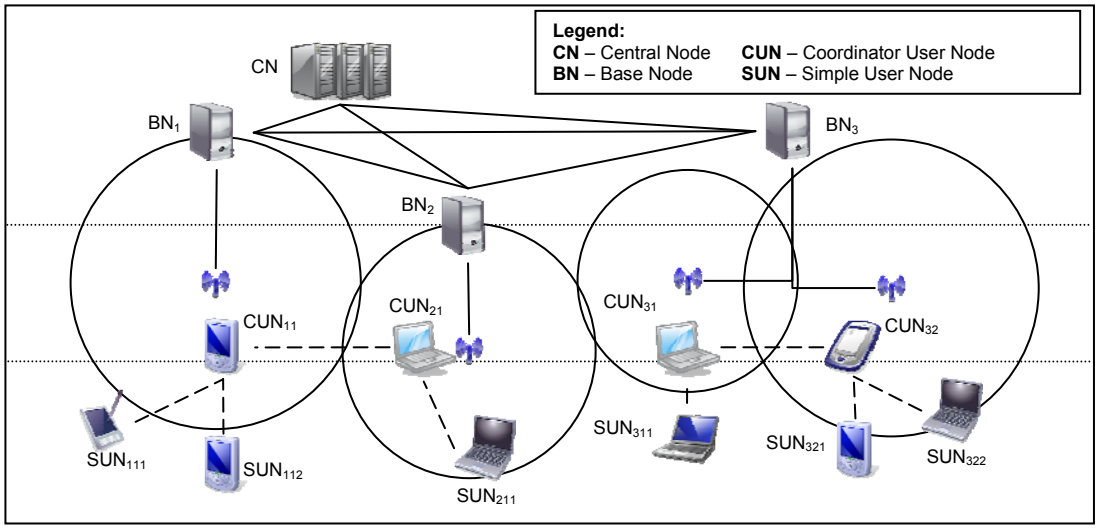

Figure 7.1. SALES Distributed Architecture.

from the uppermost to the lowest one, we consider:

Central Node (CN) - The tree root is a logical centralized and fixed entity useful to ensure context data persistency and availability of system-wide visible data. Hence, this node enables the context data distribution with the widest possible distribution scope. Since the number of received requests could be high, the CN can be realized by means of clustered architectures to enhance scalability and reliability. Finally, it can be accessed only by the nodes belonging to the level below, that usually communicate with it through high-performance fixed network connections.

Base Node (BN) - A BN is connected to and manages access network elements of heterogeneous wireless fixed infrastructures (e.g., WiFi APs, 3G/4G cellular base stations, …), and takes care of context data/query routing to/from the mobile nodes available at the level below. Each BN defines a reduced distribution scope, and can communicate only with the CN, its own neighbours, and served mobile nodes. Finally, since a BN is a full-fledged physical server, we expect it to memorize context data in order to reduce the requests relayed to the CN.

Coordinator User Node (CUN) - Mobile nodes are organized in clusters to build smaller distribution scopes. In each cluster, we dynamically elect a cluster-head, namely a CUN, useful to better control the context data distribution and to bridge together ad-hoc and infrastructure-based networks. CUNs exchange context data with close mobile devices through ad-hoc links, thus reducing the number of requests relayed to upper levels. Finally, each CUN executes proper mobility management protocols to associate with the BN in charge of the current physical place, so to connect to SALES fixed infrastructure.

Simple User Node (SUN) - Each mobile node, that is not a CUN, plays the role of a

SUN. Similarly to CUNs, SUNs enact as context source/sink into the system by injecting and requiring context data. They communicate with close mobile devices, either SUNs or CUNs, through ad-hoc links. To access SALES CDDI, each SUN has to associate with a reachable CUN; hence, proper mobility management protocols are also executed to let SUNs discover and associate with one of the CUNs available in the physical proximity.

In conclusion, the adopted distributed architecture connects and bridges together a fixed and a mobile infrastructure to increase system scalability. An extremely appealing and difficult to achieve goal is to handle most of the context distribution process through ad-hoc links; however, this clashes with both the limited network resources and the limited visibility scopes ensured by ad-hoc communications. Hence, the intervention of the fixed infrastructure is required to both ensure context data availability and perform context processing operations.

7.3. Context Data Management Layer

Our SALES CDDI addresses context data distribution in hybrid network deployments. As stated before, the availability of a fixed infrastructure simplifies the design and the realization of particular management facilities; in addition, it enables hybrid solutions where the mobile and the fixed infrastructures cooperate together toward the common goal of context data distribution. In the remainder, we discuss the main solutions adopted by SALES at each facility contained into the context data management layer (see Section 4.2 for an in-depth presentation of this layer).

Starting with context data representation, similarly to RECOWER, SALES adopts an object-oriented approach [55]. Leaving out the attributes used to describe type-specific context aspects, each context data instance has five management parameters. *Source ID (SID)*, *Version Number (VN)*, *Foreseen Lifetime (FL)*, and *Remaining Lifetime (RL)* parameters are the same ones introduced in RECOWER CDDI (see Section 6.3); in addition, *Hierarchical Level Tag (HLT)* parameter is useful to limit instance visibility into the SALES distributed architecture, for instance, to keep context data only on the mobile infrastructure. Finally, as regards QoC-based data management, SALES can tag each context data instance with additional quality metadata, such as precision and resolution.

Focusing on context data storage, SALES memorizes context data both on mobile devices and on fixed servers. Although the memorization overhead can be very high, the fixed infrastructure can be effectively used to offload context data; at the same time, context data caching on the mobile infrastructure is appealing since it can reduce context

retrieval times and improve system scalability. Similarly to RECOWER, SALES adopts distributed data caching solutions: each mobile node has a local repository of context data, with a maximum size D_{MAX}, shared with close neighbours. However, here we aim to better inspect main requirements and solutions, and compare multiple different caching algorithms to find good tradeoffs for data caching at both infrastructure and mobile trunks. Mobile nodes can freely roam, and can experience different access patterns according to the current physical location; hence, they must be able to quickly adapt, so as to improve cache usefulness under time-varying access patterns. The main management operation that differentiates caching policies is the replacement algorithm, namely the function that, when the cache is full, selects the data instance to delete to make room for the incoming data. For the sake of completeness, in Section 7.3.1, we briefly discuss the most important caching approaches in literature, by also clarifying their main shortcomings; then, in Section 7.3.2, we present our solution, called Adaptive Context-aware Data Caching (ACDC), that exploits information coming from access patterns and data instance replication into the physical neighbourhood to select the element to evict.

Finally, moving to the context data processing facility, SALES CDDI only offers very simple solutions to perform context data aggregation and filtering. We remark that the availability of a fixed infrastructure simplifies the introduction of aggregation and filtering operators: in fact, heavy computations can be dynamically offloaded to BNs that, by having full access to context data instances, can perform needed computations and send results back to mobile nodes. Also, SALES does not currently address context data confidentiality, integrity, and availability, although they are fundamental in real-world deployment scenarios. Let us remark that we did not consider such aspects since out-of-scope in respect of this thesis work.

7.3.1. Data Caching Algorithms

Above all, First In-First Out (FIFO), Least Frequently Used (LFU), and Least Recently Used (LRU) are common caching algorithms based on very simple replacement policies, so to reduce cache management overhead. FIFO orders data according to their insertion: when a data instance has to be inserted and the cache is full, the oldest element is deleted. Since cache accesses do not result in data reordering, FIFO implementation is very fast, but it does not make any effort to keep most accessed data. LFU exploits data access frequencies: for any data, it stores a counter of performed accesses, and most accessed data are maintained into the cache; since cached data are ordered according to

frequency counter values, accesses lead to dynamic data reordering. The main LFU advantage is that it maintains a cumulative view of the history of accesses: if the access pattern is static and biased, LFU adapts itself to grant the maximum number of local hits. However, since it does not quickly adapt to time-varying accesses patterns due to history effects, it can end up by storing data not useful anymore, thus leading to reduced performance. Finally, LRU dynamically reorders cached data according to most recent access times; if the cache is full, the least recently used data is deleted. Data are dynamically reordered: if a data is accessed, it is moved to the head of the cache, while the tail points to the first data to remove. LRU is simple and adapts to data accesses: unfortunately, it can cache instances that are unlikely to be accessed again (e.g., instances accessed only once and never accessed again).

FIFO, LFU, and LRU are very suitable for mobile scenarios as they introduce a limited overhead that, at the same time, allows good scalability when the cache size increases. However, in our main scenario, caching algorithms do not have strict execution deadlines, and can also introduce longer access and replacement times. We think it is more convenient to spend longer time during cache accesses and replacements than wasting network bandwidth for additional data distributions due to cache misuse. Consequently, we are interested in more complex cache replacement policies capable of increasing cache usefulness.

Following that direction, different collaborative data caching approaches in MANETs have been proposed in literature. In [113], authors present a collaborative cluster-based data caching approach. Each mobile node divides its own cache in a private and a shared area to store data of interest to, respectively, the node itself and other cluster members; the cluster-head selects the data to be moved from the private to the shared area, while LRU is used as replacement policy in each area. A close work proposes a collaborative caching framework where each node can cache either data or paths towards the data [114]; the decision of caching either data or data paths is based on the hop distance from the data: for close data, data path caching is preferred to reduce the total number of replicas in physical proximity. In addition, [114] employs LFU to select the element to evict when either the data cache or the data path cache is full. Zone Cooperative (ZC) caching builds one-hop clusters in which cooperative data caching is used: ZC uses a replacement policy based on performed accesses, hop distance from source, data lifetime and size, to select the element to evict [115]. Hence, to increase cache diversity between close mobile nodes, it uses a replacement policy based on hop count. Finally, Group-based Cooperative Caching

(GroCoCa) is a data caching solution for wireless broadcast environments. GroCoCa aims to group nodes with similar context interests and mobility patters, and exploits those clusters to perform cooperative caching [58]. This approach is definitely an interesting one, but it requires the availability of GPS localization system to properly drive cluster formation.

To conclude, although the aforementioned caching approaches are extremely valid solutions, none of them satisfies our three main requirements. First, since context data have a limited lifetime, caching approaches for CDDIs have to consider it to prevent the storage of soon-to-expire data. Second, since mobile nodes can experience time-varying access patterns consequence of physical/logical locality with close neighbours, caching approaches for CDDIs have to quickly adapt, so as to prevent the storage of data not useful anymore. Finally, traditional proposals do not usually exploit visibility of data cached on neighbours; for instance, they can inefficiently eliminate a data instance with only one copy to maintain another one with several replicas in the physical proximity. Hence, to address all those requirements, we designed our novel ACDC caching algorithm.

7.3.2. Adaptive Context-aware Data Caching

ACDC has both a local and a distributed nature, and we claim the need of both perspectives. About the local part (local ranking), ACDC strives to adaptively tailor data ranking depending on current access pattern, so to better fit current situation and reduce relayed queries. ACDC maintains a limited history (H) of data access times, and combines 1) the access frequency in the limited time-frame represented by H, and 2) data remaining lifetime, to quickly self-adapt cache when access patterns change. As regards the distributed part (remote ranking), ACDC aims at increasing the probability of retrieving needed data in a neighbour node. In particular, to increment the number of data cached in the same physical locality, ACDC controls the number of data replicas, and adopts reactive replication to store useful context data on underutilized neighbours. Finally, ACDC melts together local and remote rankings to associate each data with a final utility value used to select, when necessary, the element to remove.

With finer details, and starting from local ranking, we foresee two borderline types of significant access patterns: uniform and preferential accesses. In uniform access patterns, each data has almost the same probability of being reclaimed in the future while, in preferential ones, some data are more requested than others. Both these access patterns strictly relate with locality principles: if there is strong locality, either physical or logical,

between nodes in the same area, queries will match similar data, thus resulting in preferential accesses; otherwise, nodes tend to emit queries with different set of matching data, thus resulting in uniform accesses. Different access patterns modify the utility of cached data, hence, it is important to estimate the current access distribution: since uniform accesses do not allow future accesses forecasting, it is advisable to maintain data with higher probability of being asked before their expiration, namely data with longer lifetime. On the opposite, as preferential accesses allow a more accurate forecasting of future accesses, it is advisable to preserve data with higher probability of being required, namely more frequently used data.

Toward data access pattern estimation, ACDC calculates the linear correlation (named correlation index in the remainder) between 1) the time spent by the data into the cache according to H; and 2) the number of accesses registered in H. For uniform access patterns, the history of the accesses registered by H will be quite random and will not highlight any relationship between the two above indicators, thus leading to lower linear correlation values. Instead, for preferential access patterns, the two indicators will present a higher linear correlation, due to the fact that context data kept in cache for longer period will be also the ones with higher number of accesses. The correlation index is evaluated over H, and we have to consider that H length is useful to trade off accuracy with adaptation promptness. In fact, while roaming, a mobile node reaches different locations with different neighbours and potentially different interests. Since long histories tend to melt together access patterns belonging to different situations, they hinder the usefulness of forecasting and also slow down adaptation mechanisms; hence, ACDC uses a short history H to quickly adapt to the current situation. Once evaluated the correlation index, ACDC uses it as weighting factor for the local ranking: for uniform access patterns, it favours data with longer lifetime while, for preferential ones, it favours data more frequently accessed in H.

Focusing on remote ranking, we remark the importance of controlling the number of data copies in the neighbourhood, so to increase the total number of different data available in the physical area. In ACDC, each node periodically disseminates to its one-hop neighbourhood lightweight summaries of its cache; in particular, each neighbour cache summary contains the number of cached data, maximum cache size, and a compact representation of cached data. Thanks to those summaries, each node can locally estimate a remote rank based on the number of replicas stored in the neighbourhood: the higher the number of replicas of one data, the higher the probability that a copy will be removed.

To select the data to remove, ACDC melts together local and remote rank values and computes a utility value for each cached data. In addition, ACDC reactively replicates data with high utility value: in fact, it could be the case that the node has to remove an important data due to space constraints; hence, ACDC strives to replicate it on a neighbour node, to keep it available for future requests. However, greedy replication can introduce interferences with near nodes. If a node greedily replicates its data in one neighbour, neighbour cache will be no longer related to past queries, thus increasing the probability of not retrieving useful data into the cache. Hence, to select the neighbour to replicate the data on it, ACDC considers only neighbours with a small ratio between the current cached data and the maximum cache size, so to avoid excessive neighbour perturbation.

7.4. Context Data Delivery Layer

The context data delivery layer of SALES shares similarities with the one of RECOWER, but extends it to support hybrid scenarios with fixed wireless infrastructures. SALES adopts a subscription flooding approach that exploits an incremental search into the distributed hierarchical architecture, with the main goal of retrieving required context data as close as possible to the query sender node in order to reduce management overhead. Following our guidelines, SALES first tries to find data on lower hierarchy levels; then, in case of not positive response, it incrementally routes the query to the upper levels.

SALES context data routing is also based on context queries. Context data are distributed only as consequence of matching context queries, that trigger distributions from remote data repositories toward the query creator node. Context queries build temporary routing paths into the distributed architecture that, if required, can also reach the fixed infrastructure. Since each node can communicate only with its father node, neighbours, and served nodes, SALES context data distribution can exploit different dissemination scopes of increasing sizes to enforce physical locality principle.

For the sake of clarity, we now present a brief example of context data distribution in SALES. By default, both data and queries are distributed along the vertical path between the data/query creator node and the CN (SUN_{211} propagates the new produced data up to the CN, step 1 in Figure 7.2, solid red arrows). This vertical distribution is useful to both increase data/query visibility (up to the whole distributed system) and trigger the matching phase with data/queries available on intermediate nodes. However, to increase the probability of finding context data in lower hierarchical levels, so to reduce the traffic on

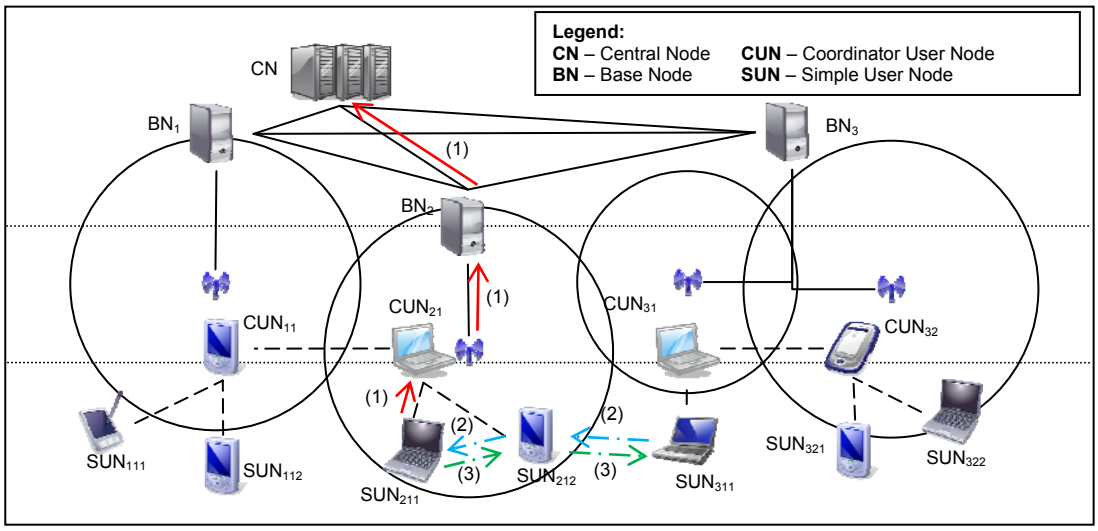

Figure 7.2. Example of SALES Context Data Distribution.

the fixed infrastructure, context queries are also horizontally distributed at the same hierarchical level. For instance, in Figure 7.2 (step 2, dashed arrows), SUN_{311} obtains the required data from SUN_{211}; it emits a query that, through SUN_{212}, reaches SUN_{211}. This horizontal distribution is justified when the requesting node is looking for context data strictly related with the current physical place, such as place profiles, since they are likely to be available on neighbours in physical proximity. Then, SALES performs data routing on a hop-by-hop basis by always involving single steps into the distributed architecture. When a positive data/query match occurs on a node, SALES generates a context response and routes it back to the node that had relayed the query (in Figure 7.2 (step 3), this leads to a final data path SUN_{211}-SUN_{212}-SUN_{311}).

SALES exploits QoC parameters to adapt the routing process. In particular, as clarified in Section 7.4.1, it exploits QoC data retrieval time to reconfigure the maximum routing delays at each intermediate node. In addition, since resource management is fundamental as mobile users would not accept fast battery depletion and heavy management overhead, SALES automatically adapts query processing rates to limit CPU load; Section 7.4.2 presents how our CDDI automatically drops context queries that would lead to heavy CPU management load.

7.4.1. Data Retrieval Time Enforcement

Between different quality attributes, context-aware services can specify a QoC data retrieval time, namely the maximum time between context query emission and context data delivery to the mobile node. By exploiting this attribute, SALES can adapt at runtime to introduce appropriate routing delays depending on current available resources, while

always enforcing service QoC constraints. These routing delays are fundamental in relieving a congested network, so to prevent wireless storm issues [105]. In addition, as better detailed in Section 7.5.2, they enable the introduction of batching techniques, namely all those solutions that aim to reduce the number of physical transmissions by grouping many short messages in a big one.

To implement the proposed quality-based context distribution process, each context query contains seven management parameters; here, for the sake of clarity, we also recall the query parameters presented in RECOWER, and we extend them to consider hybrid network deployments. *Horizontal Time To Live (HTTL)* is the maximum number of nodes traversed at the same hierarchy level, and is useful to limit query visibility on both the mobile and the fixed infrastructure. *Maximum Query Response (MQR)* is the maximum number of data instances collected by this query, and is mainly used to prevent excessive data retransmissions by anticipating query removal. *Query Routing Delay (QRD)* and *Data Routing Delay (DRD)* represent the delays each node can apply to query/data before routing them to the next hop; as presented in Section 7.5.2, they are fundamental to enable batching techniques in SALES. *Already Collected Data (ACD)* contains the list of the keys associated with already routed data, and is fundamental to prevent useless data retransmissions during collection. *Query Level Mask (QLM)* limits the vertical visibility of the context query, for instance, to keep it only on the mobile infrastructure and up to CUN nodes; that allows to better trade off introduced management overhead, especially when the fixed infrastructure is overloaded. Finally, *Query LifeTime (QLT)* is the maximum absolute lifetime of the query, and is used to mark query expiration and removal.

If a mobile node, either CUN or SUN, seeks for specific context data, it builds and emits a proper context query matching them. The query contains the data filter used to select matching data; similarly to what we did in RECOWER, the data filter is represented by a set of constraints on data attributes, arranged by AND/OR functions. Before query distribution, proper management parameters have to be chosen to ensure agreed data retrieval time. To simplify context-aware services development, SALES automatically maps the required data retrieval time to the associated query parameters. In the following, we present the general mapping process between data retrieval time and query parameters. For now, we assume HTTL defined either by the service level or by the quality contract associated with the sender node.

Above all, SALES has to compute both QRD and DRD, namely two parameters that, together with HTTL, deeply influence the whole routing process. The incremental context

data search does not distribute data/queries immediately, but introduces local routing delays to manage the distribution process and to avoid useless distributions when context data are supplied by neighbours belonging to the same hierarchical level. Since it is impossible to know which nodes cache matching context data, all the subsequent considerations are based on the worst-case scenario where the query has to reach the CN before finding matching data.

In finer details, the evaluation of DRD and QRD is based on several considerations. First, each node involved into the routing process introduces a maximum delay of QRD in query distribution and a maximum delay of DRD in data distribution: hence, a maximum total delay of (QRD + DRD) for each additional hop in the routing process. Second, before relaying the query to the upper level, each node belonging to the vertical path between the query creator node and the CN waits a total time of (HTTL × (QRD + DRD)) to let close peers route possibly matching data. If query HTTL is zero, no horizontal distribution is performed; hence, the query is simply relayed to the upper level after a total delay of QRD (to consider these different contributions, H is a binary variable equal to 1 if query HTTL is bigger than zero, 0 otherwise). Third, to always ensure agreed data retrieval time, we have to consider that, in the worst-case scenario, SUNs experience longer routing times than CUNs since farther from the CN. Hence, both DRD and QRD evaluation must depend on the level in the hierarchy of the node that emits the query (S is a binary variable equal to 1 if the node is a SUN, 0 otherwise). Finally, all the delays obtained through a simple mathematical mapping do not consider unwanted, unforeseen and not measurable delays due to operating system multi-tasking, limited bandwidth, and so forth. Obtained delays are ideal and, in the following, we use the subscript M for DRD and QRD to suggest that they both represent maximum (M) nominal times. Putting all together, formula (7.1) represents the worst-case time needed to propagate the query up to the CN. After query distribution, matching context data have to be vertically routed to the query sender node. Formula (7.2) is the worst-case time required for data distribution from the CN to the query sender node; of course, it considers that SUNs experience longer delays than CUNs due to higher distance from tree root.

To conclude, directly descending from SALES context data/query distribution (see Figure 7.2), the maximum data retrieval time and HTTL/QRD/DRD are related by the subsequent formulas (7.1)-(7.3):

$$\text{Query distribution time} = \begin{cases} \textit{(BN Level)} & H \times HTTL \times (QRD_M + DRD_M) + (1 - H) \times QRD_M + \\ \textit{(CUN Level)} & H \times HTTL \times (QRD_M + DRD_M) + (1 - H) \times QRD_M + \\ \textit{(SUN Level)} & S \times (H \times HTTL \times (QRD_M + DRD_M) + (1 - H) \times QRD_M) \end{cases} \quad (7.1)$$

$$\text{Data distribution time} = (2 + S) \times DRD_M \quad (7.2)$$

$$\text{Data Retrieval Time} = \text{Query distribution time} + \text{Data distribution time} \quad (7.3)$$

Hence, given a particular data retrieval time, SALES can apply above formulas to find DRD_M and QRD_M. However, formulas (7.1)-(7.3) form an undetermined system with infinite solutions. To find a feasible solution, we relate DRD_M and QRD_M with the additional constraint expressed in formula (7.4):

$$DRD_M = \gamma \times QRD_M \quad (7.4)$$

where $\gamma \geq 1$ to favourite data routing adaptation. In fact, data transmissions are usually more frequent than query ones, and higher γ values increase the possibility of adapting context data routing, for instance, to avoid retransmitting the same context data in a small time frame or delaying such transmission if the wireless channel is very busy. Finally, to have a time margin useful to recover unforeseen runtime delays, SALES introduces a weighting factor α ($\alpha < 1$). Hence, the final DRD and QRD, carried by a query, are obtained from DRD_M and QRD_M by means of formulas (7.5)-(7.6):

$$DRD = \alpha \times DRD_M \quad (7.5)$$

$$QRD = \alpha \times QRD_M \quad (7.6)$$

Formulas (7.1)-(7.6) let SALES automatically derive a suitable pair of DRD and QRD delays that ensure agreed data retrieval time. The weighting factor α can be either statically or dynamically defined, so to account for delays introduced by real-world systems. We note that the correct sizing of such parameter is not straightforward as it actually depends on runtime conditions, such as mobile node load status. Hence, since the dynamic evaluation of α is not easy to be addressed with low management overhead, we assume α statically set depending on system scale and predictions over the expected maximum system load.

7.4.2. CPU-aware Context Query Processing

SALES context routing relies upon context queries to efficiently route context data into the system. Considering that mobile devices have limited resources in terms of CPU, memory, and battery, SALES introduces additional mechanisms to control and keep the introduced management overhead as low as possible. Above all, context query processing is the first responsible of the CPU load introduced on mobile nodes: in fact, queries have

to be matched with locally stored data and, if permitted by associated parameters, distributed again to peers and/or father nodes. In addition, they can trigger data distributions, so further increasing local CPU load.

Hence, to control the CPU overhead introduced by SALES, we need to limit the number of processed queries for time period. Unfortunately, by better analyzing SALES context distribution process, we remark that the number of queries processed by a mobile node depends on three main factors: 1) node density; 2) hierarchy level; and 3) data access patterns. In fact, if the mobile node is in a high density area, it will probably receive more queries than if it would have been in a low density one. In addition, if the mobile node is a CUN in charge of routing data/queries on behalf of served SUNs, it will probably experience increased CPU load due to additional management duties. Finally, if the mobile node already stores required data, it can answer right away, thus experiencing a reduced CPU load; otherwise, it has to distribute the query to neighbours, and, perhaps, to upper level, thus experiencing a higher CPU load.

Consequently, the precise estimation of the CPU load introduced by SALES at runtime would require a complex model based on several time-varying and unpredictable aspects. Monitoring and processing all such aspects would probably introduce an unfeasible overhead on resource-constrained mobile devices. Hence, we adopted a more lightweight solution that, even if less precise, can run on traditional mobile devices with contained overhead.

From a general viewpoint, a first solution, called "naïve query drop" in the remainder, exploits a sliding window over last processed queries and a rigid threshold to reduce the number of queries processed in a particular time period. Given a static threshold PQ_{MAX}, this policy ensures that a maximum of PQ_{MAX} queries are processed in each period, e.g., each second. Toward this goal, each node has a limited history of timestamps, called H_{TS}, representing the times associated with the last received and processed queries (see formula (7.7)).

$$H_{TS} = \{TS(1), TS(2), ..., TS(i)\}, TS(1) \leq ... \leq TS(i), i \leq PQ_{MAX} \quad (7.7)$$

When a new query arrives, this policy first defines a new history H_{TS}' from H_{TS}, as presented below in (7.8)-(7.11).

$$z = \min(i+1, PQ_{MAX}) \quad (7.8)$$

$$H_{TS}' = \{TS'(1), TS'(2), ..., TS'(z)\} \quad (7.9)$$

$$TS'(z) = Now \quad (7.10)$$

$$TS'(j) = TS(j+1),\ 1 \leq j \leq z\text{-}1 \tag{7.11}$$

$$totalQueries = (TS'(z) - TS'(1)) \times PQ_{MAX} \tag{7.12}$$

Then, it checks if H_{TS}' respects PQ_{MAX}: it considers the time period between the first and the last element, computes the maximum number of queries that can be processed in this period respecting PQ_{MAX} (see formula (7.12)), and checks that this value is not lower than the number of elements contained in H_{TS}'. In that case, the new query is accepted and the history H_{TS}' is assumed to be the new H_{TS}; otherwise, the query is dropped and H_{TS} is not updated. Hence, H_{TS} is progressively shifted to keep TSs of the last PQ_{MAX} processed queries.

Although this policy is effective in reducing the queries processed in a time period, it has few important shortcomings. First, since it does not consider any external feedback associated with the real CPU load, it can lead to CPU misuse. Second, PQ_{MAX} needs to be known a-priori, but this is a strong assumption in heterogeneous environments where devices can have different PQ_{MAX} values. Third, fixing a rigid threshold on processed queries makes sense only if we can correctly estimate the CPU load introduced by each processed query but, as explained before, that is not possible due to the many intertwined aspects that influence the distribution process. Finally, it assumes that PQ_{MAX} is static, but this could be not the case if data access patterns change over time.

To overcome such limitations, SALES introduces an adaptive policy, called “adaptive query drop”, that dynamically adjusts PQ_{MAX} depending on feedbacks coming from both CPU load and context data distribution process. Since this policy introduces runtime adaptation features, it will be presented in Section 7.5, devoted to the Runtime Adaptation Support.

7.5. Runtime Adaptation Support

SALES adapts its own runtime behaviour according to working conditions. Adaptation mechanisms affect both the context data management and the delivery layer, with the main goal of reducing introduced overhead for the sake of system scalability. This section presents finer details associated with SALES adaptive mechanisms: in Section 7.5.1, we discuss data caching adaptation; then, in Section 7.5.2, we introduce the different transmission policies offered by our CDDI, with a specific focus on the adaptive variant; finally, in Section 7.5.3, we introduce details on the adaptive query drop policy, useful to control CDDI CPU load.

7.5.1. Adaptive Context Data Caching

As presented in Section 7.3.2, ACDC exploits a replacement algorithm made by both a local and a remote ranking component. Here, we focus first on local ranking by presenting how ACDC evaluates the correlation index and uses it to calculate local score; then, we present remote ranking by introducing details on the estimation of data instance replication; finally, we clarify how ACDC merges such indicators to find the utility values used by replacement. For the sake of clarity, Figure 7.3 shows ACDC pseudo-code.

Let us focus on local ranking. Starting with the linear correlation index, ACDC exploits the Pearson product-moment correlation coefficient [116]. Each time a new query is received (function *receiveQuery* in Figure 7.3), all cached data are matched with query filter. For each positive data/query match, the function *recordAccessDescriptor* updates the limited history H with the new access descriptor; then, *scheduleSendData* generates and sends the new context data response. The evaluation of the Pearson coefficient is periodically triggered and is based on two vectors, namely X and Y, used to store the values to be correlated. In particular, when the correlation index needs to be updated, ACDC computes X_i and Y_i ($i \in [0; CacheMaxSize-1]$) for all the data in the cache as follows: X_i is the period between the newest and the oldest access descriptor in H

Variables
- C: local cache repository
- C[i]: i^{th} data in the local cache
- C_CurrentSize: local cache current size
- C_MaxSize: local cache maximum size
- N: set of node current neighbors
- N_size: size of the node current neighbors
- H: accesses history
- H_CurrentSize: current history length
- H_MaxSize: maximum history length
- NeighCacheSummary: map of repository status for N
- NeighCacheSummary[n]: repository status for the node n
- correlationIndex: current correlation index value

Functions
- **piggybackOnMobilityBeacon(m):** piggyback message m in the next mobility beacon sent to all 1-hop neighbors
- **scheduleSendData(d,n):** schedule to send data d to node n
- **storeQuery(q):** store a query q in local repository

Messages
- **REPOSITORY_STATUS**<C_CurrentSize, C_MaxSize, f>: message contained repository status
- **QUERY** <q>: message used to distribute query q

```
Received msg QUERY<q> from node n.
receiveQuery(n, q)
1: for all d ∈ C do
2:    if (q.match(d)) then
3:       recordAccessDescriptor(d);
4:       scheduleSendData(d, n);
5:    if (!q.isValid())
6:       break;
7: if (q.isValid()) then
8:    storeQuery(q);

recordAccessDescriptor (d)
1: if (H_CurrentSize >= H_MaxSize) then
2:    H.removeOldestElement();
3:    H_CurrentSize--;
4: H.add(Now, d);
5: H_CurrentSize++;

Invocked every beacon period
sendNeighCacheSummary()
1: Build an empty Bloom filter f
2: Build m = REPOSITORY_STATUS< C_CurrentSize, C_MaxSize, f >
3: for all d ∈ C do
4:    f.add(d.key);
5: piggybackOnMobilityBeacon(m)

Received msg REPOSITORY_STATUS<C_CurrentSize, C_MaxSize, f>
     from node n
receivedNeighCacheSummary (n, C_CurrentSize, C_MaxSize, f )
1: NeighCacheSummary[n] = < C_CurrentSize, C_MaxSize, f >

Reclaimed when a new data arrives with cache full.
evictLessValuableData()
2: for all d ∈ C do
3:    d.rank = 0.4 × localRank(d) + 0.6 × remoteRank(d)
4: dataToEvict = the data with the minimum rank
5: lessLoadedNode = null;
6: if (dataToEvict.rank > 0.5 && remoteRank(d) >= 0.7) then
7:    for all n ∈ N do
8:       cacheLoadFactor = n.C_CurrentSize/n.C_MaxSize;
9:       if (cacheLoadFactor< 0.5 && cacheLoadFactor<
            lessLoadedNode.cacheLoadFactor &&
               !NeighCacheSummary[n].f.contains(d.key)) then
10:         lessLoadedNode = n;
11: if (lessLoadedNode != null) then
12:    scheduleSendData(dataToEvict, lessLoadedNode);

localRank (d)
1: lifetimeComponent = d.RL/d.FL;
2: accessRatioComponent = d.accessInH/maxAccessesInH;
3: return correlationIndex × accessRatioComponent + (1-
     correlationIndex) × lifetimeComponent;

remoteRank (d)
4: count = 0;
5: for all n ∈ N do
6:    if (NeighCacheSummary[n].f.contains(d.key)) then
7:       count++;
8: return 1 - count/N_size;
```

Figure 7.3 Pseudo-code of the ACDC Replacement Policy.

associated with the current data instance, while Y_i is the cumulative number of accesses to the data in H. ACDC uses these two vectors to obtain the Pearson coefficient by using the formula (7.13) ($\overline{X}$ and $\overline{Y}$ respectively represent the average value of X and Y):

$$PearsonIndex = \frac{\sum_{i=1}^{n}(X_i - \bar{X}) \times (Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2} \times \sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \quad (7.13)$$

$$correlationIndex = (PearsonIndex > 0)\ ?\ PearsonIndex : 0 \quad (7.14)$$

By construction, the Pearson coefficient is in [-1; 1]: the more X and Y are linearly correlated, the closer to one (in absolute value) the coefficient becomes. The sign allows to distinguish whether the two variables are either positively or negatively correlated, namely whether an increment on one variable results either in an increment or in a decrement of the other one. During preferential access patterns, X and Y are positively correlated, hence, in that case the Pearson coefficient tends to 1. Instead, during uniform access patterns, X and Y are weakly correlated, and the Pearson coefficient tends to 0. In our case, only positive values of the Pearson coefficient are useful; negative ones are related to access patterns variation and data replacement, and do not allow efficient forecasting. In conclusion, the final correlation index considered by ACDC to compute the local rank, called *correlationIndex* in Figure 7.3 and obtained by formula (7.14), is equal to the Pearson coefficient if positive, or 0 otherwise.

In particular, for each data, ACDC local rank merges together its *lifetimeComponent*, i.e., the ratio between data RL and FL, and its *accessRatioComponent*, i.e., the ratio between the accesses a specific data has in history H and the maximum data accesses value for all data, as expressed by formula (7.15):

$$(1\text{- correlationIndex}) \times \text{lifetimeComponent} + \text{correlationIndex} \times \text{accessRatioComponent} \quad (7.15)$$

That combination permits to weight more data instances either with longer lifetime for equally distributed access periods, or more frequently accessed during preferential access periods.

Considering remote ranking, ACDC disseminates neighbour cache summaries by piggybacking them in mobility beacons periodically sent by SALES to manage node mobility. The function *receivedNeighCacheSummary* (see Figure 7.3) is invoked when a new cache summary is received, and the *NeighCacheSummary* stores repository statuses for neighbours, indexed by node identifier. The function *sendNeighCacheSummary* creates the summary associated with the local repository and adds it to the mobility beacon sent to one-hop neighbours. Then, the remote rank is calculated as expressed in formula (7.16):

$$1 - \frac{\text{Number of Neighbours already Storing the Data}}{\text{Total Number Of Neighbours}} \quad (7.16)$$

Finally, ACDC combines the local and remote rank in a final utility value used to order cached data. As it is advisable to save less replicated data, the local rank (function *localRank*) and the remote rank (function *remoteRank*) are combined with different weights (respectively 0.4 and 0.6). That reduces local ranking effects; in case of similar local values, the remote rank allows to decide the data instance to save.

After data removal, the function *evictLessValuableData* can decide to perform reactive replication. The replication process is triggered every time that a locally cached data with a utility value higher than 0.5 has to be removed. To prevent high replication degree, the process takes place only if the data instance is cached by less than 30% of neighbours. If this is the case, the node selects the less loaded neighbour that does not already store the data and with a ratio between current and maximum size lower than 0.5 (see Figure 7.3), and replicates the data there. In this way, if that data instance is queried in the future, it will be likely found in the neighbourhood.

7.5.2. Data and Query Transmission Policies

As clarified in Section 7.4.1, SALES introduces proper data/query routing delays (DRD/QRD) to enable context distribution process adaptation; among several advantages, our CDDI exploits them to implement batching techniques. Broadly speaking, a batching technique aims to queue multiple data/query to send them in a unique message, so as to avoid multiple channel accesses and reduce management overhead consequence of packet headers. Although batching techniques are useful to increase system scalability, queuing delays have to be carefully controlled to avoid excessive QoC degradation, for instance, consequence of the delivery of context data not related anymore with the current situation. Hence, SALES introduces several batching techniques with different tradeoffs between delivery timeliness and management overhead. In particular, it implements an adaptive solution that dynamically adjusts the data retrieval time according to the current situation: if the wireless network is far away from congestion, it anticipates data/query distributions to offer better quality; instead, if the wireless network is close to congestion, it ensures only necessary quality requirements. Now, we better detail the different data/query transmission policies offered by our SALES CDDI, starting with the simpler one.

During context data/query routing, each node exploits query parameters to schedule further distributions. Both DRD and QRD identify precise time instants in which the

associated data/query has to be distributed, in order to respect the overall QoC data retrieval time. Hence, a first simple distribution policy, named "no batching", transmits data/queries as soon as their own distribution time (respectively imposed by DRD/QRD) is reached. However, this policy performs one transmission, hence one wireless channel access, for each data/query to be transmitted, thus potentially leading to a high number of conflicts in wireless channel accesses that, in their turn, trigger MAC backoff mechanisms.

To reduce wireless channel contention, data/query distributions have to be scheduled in time periods; in particular, we define *distribution period* as the *interval in which a data/query can be transmitted without compromising agreed data retrieval time*. If data/queries have to be transmitted in periods, and not precise time instants, SALES can select the final transmission times to batch more data/queries in the same message. Following this direction, two additional policies, called "naïve batching" and "adaptive batching", strive to reduce wireless channel accesses. Main difference between them is that adaptive batching adapts distribution periods depending on wireless channel congestion: during not congested situations, tighter distribution periods are used to ensure lower retrieval times; instead, during congested periods, larger distribution periods let SALES batch more data to reduce wireless network accesses.

In finer details, considering the mapping introduced in Section 7.4.1, DRD and QRD are lower bounds that must be respected to avoid early distributions, while DRD_M and QRD_M are upper bounds that must be respected to ensure agreed data retrieval time in an ideal situation, namely no delays due to local processing and data transmission. Hence, in the naïve batching, SALES automatically selects data and query transmission time respectively within the [DRD; DRD_M] and [QRD; QRD_M] distribution periods. Naïve batching adopts DRD_M/QRD_M as distribution upper bounds to trigger the real data/query exchange. Any time a data/query distribution reaches its own transmission upper bound, SALES finds all the data/query distributions for the same destination, and whose distribution period contains the upper bound of the current processed distribution. Then, it batches and transmits them together in a unique message, thus reducing the total number of wireless channel accesses.

Let us show now benefits and shortcomings of the naïve approach with a simple example (see Figure 7.4). For the sake of simplicity, we consider the distribution of two different queries only, namely Q_1 and Q_2, that present overlapping [QRD; QRD_M] periods. Without batching (Figure 7.4 (a)), each query distribution requires one wireless channel access. With naïve batching active (Figure 7.4 (b)), when Q_2 upper bound is reached,

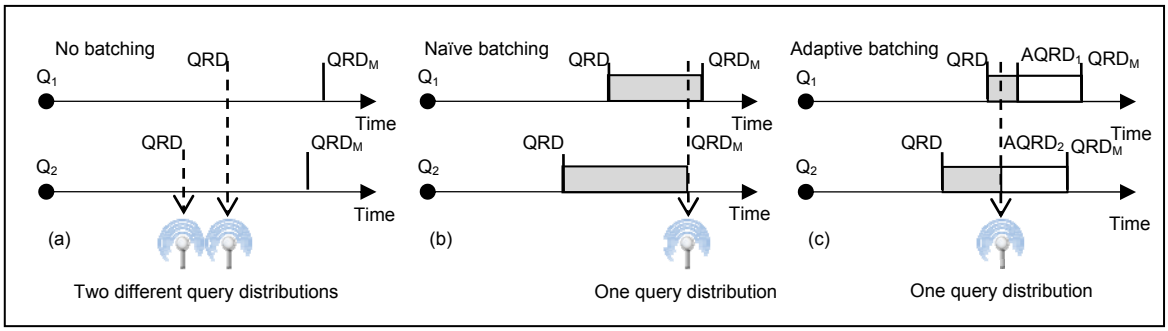

Figure 7.4. Query Distribution Example with Different Distribution Policies.

SALES detects that also Q_1 can be distributed; hence, it transmits a unique message with Q_2 and Q_1, thus saving one wireless access.

Although the naïve batching can reduce the number of wireless accesses, it presents two main limitations. First, it uses distribution period upper bounds to trigger data/query distribution: this, in its turn, introduces delays longer than necessary, and makes it more difficult to enforce agreed data retrieval time. Second, naïve batching is static: it does not monitor the current wireless network congestion, and does not adapt to time-dependent load conditions.

To solve these issues, adaptive batching introduces ADRD/AQRD (A stands for adaptive) as distribution time upper bounds (lower than DRD_M/QRD_M upper bounds), and automatically re-adjusts them by using θ, a time-dependent factor used to quantify network congestion status. θ is in [0; 1], and values closer to 1 point out a more congested network. Hence, the more the network is congested, the more the θ value is close to 1 and ADRD/AQRD are close to DRD_M/QRD_M (bigger batches and fewer wireless channel accesses), and vice versa for low θ values as shown in formulas (7.17)-(7.18).

$$\mathrm{ADRD} = \mathrm{DRD} + (\mathrm{DRD_M} - \mathrm{DRD}) \times \theta \quad (7.17)$$

$$\mathrm{AQRD} = \mathrm{QRD} + (\mathrm{QRD_M} - \mathrm{QRD}) \times \theta \quad (7.18)$$

Figure 7.4 (c) shows an associated example. Due to the adaptive approach, Q_2 has a distribution period upper bound equal to $AQRD_2$ (lower than QRD_M). When $AQRD_2$ is reached, Q_2 is distributed and Q_1 is automatically included in the same batch; SALES performs one final distribution, but with a reduced distribution time.

To estimate the current θ value, we exploit a distributed monitoring schema that contains two main phases: 1) *local load information monitoring*; and 2) *load information distribution*.

Starting from local load information monitoring, each mobile node locally estimates its own average usage of wireless channels. Each node computes a *local wireless network*

load factor, whose value is in [0; 1], to quantify the local usage of the wireless channel. Value evaluation is based on 1) an estimation of the bandwidth available on the wireless channel and 2) the outbound traffic sent by the local wireless network interface. SALES periodically estimates the bandwidth available on a particular wireless interface by a traditional active probing technique [117]. To keep the monitoring overhead as low as possible, only CUNs perform active probing, and periodically distribute estimated values to served SUNs. Even if this estimation is suboptimal due to the spatial reuse of the wireless channel, i.e., a SUN could experience wireless channel conditions different from its own CUN, it gives us a good approximation and drastically reduces the runtime monitoring overhead. Then, each node periodically executes a function to update the local wireless network load factor. First, it sets the last load factor as the ratio between the number of bytes sent in the previous period and the maximum number of bytes available on the air, equal to the product between the wireless medium available bandwidth and the time period. Then, it evaluates the current local load factor as the average of the current and the previous load factor; the average is based on equal weights, so to be able to follow fast network dynamics.

As regards load information distribution, SALES periodically distributes the estimated local load factors to let each mobile node estimate the channel usage in its own one-hop ad-hoc neighbourhood. Hence, each node periodically distributes its local network load factor in the neighbourhood, and collects load factors received from neighbours. Of course, each mobile node sends the local load factor only to the one-hop neighbours since, due to the spatial reuse of the wireless channel, it tends to interfere only with them.

Finally, each node merges the collected load factors to find a θ value able to capture the wireless channel usage in its own one-hop ad-hoc neighbourhood. Due to the adopted load factor evaluation, the final θ is equal to the sum between the local and the received load factors. This is roughly equal to the ratio between the sum of all the transmitted bytes in the one-hop ad-hoc neighbourhood and the total number of bytes that could have been transmitted in the last monitoring period. Hence, the more loaded the wireless channel, the higher the θ value will be, thus favouring the batching of an increased number of data/queries in order to reduce the number of wireless channel accesses.

7.5.3. Dynamic Adaptation of Query Processing Threshold

SALES proactively drops context queries to control introduced CPU load.

Unfortunately, the naïve drop policy has two main limitations: 1) it does not consider feedbacks coming from the real CPU load; and 2) it does not adapt to time-varying access patterns. Hence, we propose an adaptive drop policy that, by exploiting two main inputs, namely CPU load and dropped query statistics, dynamically adapts the query drop threshold. In fact, current CPU load lets us to understand if the CDDI is respecting the given maximum CPU threshold. At the same time, monitoring the number of processed and dropped queries lets us to understand if query processing rate should be adapted to increase context availability. Our adaptive proposal combines these two main inputs to dynamically reconfigure the query processing rate, so to correctly trade off introduced CPU load with context data availability.

Main goal of our adaptive drop policy is to ensure that the CDDI does not introduce a CPU load higher than a specified threshold, namely *MaxCPULoad*, while minimizing dropped queries. Since the adaptation process is triggered periodically, PQ_{MAX} value is stable between two subsequent adaptations; to remark that PQ_{MAX} varies, $PQ_{MAX}(k)$ is the value of PQ_{MAX} between the k-th and (k+1)-th adaptations.

The component that takes care of adapting $PQ_{MAX}(k)$ includes three main stages for the sake of reusability. The first one supplies useful load indicators, the second one decides if $PQ_{MAX}(k)$ has to be adapted or not, while the third one calculates and applies the adaptation step $\Delta PQ_{MAX}(k)$. If an adaptation is required, $PQ_{MAX}(k+1)$ is computed from $PQ_{MAX}(k)$ according to formula (7.19).

$$PQ_{MAX}(k+1) = PQ_{MAX}(k) + \Delta PQ_{MAX}(k) \quad (7.19)$$

Starting with the first stage, it works on a time discrete base by publishing new load indicators every monitoring period, i.e., every T seconds. It supplies three main indicators: 1) *average CPU load (ACPU(k))*; 2) *number of processed queries (PQ(k))*; and 3) *number of dropped queries (DQ(k))*. ACPU(k) gives a feedback on the real CPU load introduced by the CDDI. Since CPU load readings are usually affected by noise, we introduce a low-pass filter: we sample CPU load with a period $T_{CPU} < T$, while the final ACPU(k) is the average of the CPU load readings sampled in the previous monitoring period. PQ(k) and DQ(k) keep track, respectively, of the total number of processed queries and of dropped ones in the last monitoring period. All these values are required to prevent unneeded adaptations if the CDDI is not receiving queries at all.

Then, every monitoring period, new values of ACPU(k), PQ(k), and DQ(k) are made available to the second stage that decides if $PQ_{MAX}(k)$ has to be adapted. This stage triggers $PQ_{MAX}(k)$ adaptation in two cases: 1) if ACPU(k) is higher than MaxCPULoad, in

order to reduce $PQ_{MAX}(k)$; and 2) if ACPU(k) is lower than MaxCPULoad and DQ(k) is higher than 0, in order to increase $PQ_{MAX}(k)$ and reduce dropped queries. However, to prevent bouncing effects with $PQ_{MAX}(k)$ going above and below a particular value without stabilizing, $PQ_{MAX}(k)$ is adapted only if the current CPU load is not a transient event, such as a temporary CPU load spike. Toward this goal, we use a time series forecasting technique to predict next CPU load values. Due to its good price/performance ratio, we adopted a first-order Grey filter [118]: such a filter tries to extract signal trend from noisy values, and bases its prediction upon a limited history of sampled ACPU(k) values. History length trades off smoothness and promptness in following fast changing signals. The Grey filter gives us the predicted value of the average CPU load at the end of the next monitoring period. If this value is significantly different from MaxCPULoad, we trigger $PQ_{MAX}(k)$ adaptation; otherwise, this is probably a transient load spike and no adaptation takes place.

Finally, the third stage, if triggered, computes and applies $\Delta PQ_{MAX}(k)$. The evaluation of this adaptation step follows two main design rules. First, $\Delta PQ_{MAX}(k)$ has to be scaled in respect to $PQ_{MAX}(k)$ to avoid overreactions. Second, $\Delta PQ_{MAX}(k)$ has to assume higher values if the CDDI is introducing a CPU load higher than MaxCPULoad (or lower than MaxCPULoad with dropped queries) for long time periods to speed up the response. Consequently, if CLM is the number of subsequent CPU load misuse periods, and LoadRatio(k) is a value in [0; 1] representing if ACPU(k) is close to MaxCPULoad, $\Delta PQ_{MAX}(k)$ is retrieved as in formula (7.20):

$$\Delta PQ_{MAX}(k) = \frac{PQ_{MAX}(k)}{2} \times \text{LoadRatio}(k)^{1/CLM} \tag{7.20}$$

$$\text{CPULoadNum}(k) = \min(\text{ACPU}(k), \text{MaxCPULoad}) \tag{7.21}$$

$$\text{CPULoadDen}(k) = \max(\text{ACPU}(k), \text{MaxCPULoad}) \tag{7.22}$$

$$\text{LoadRatio}(k) = 1 - \frac{\text{CPULoadNum}(k)}{\text{CPULoadDen}(k)} \tag{7.23}$$

LoadRatio(k) is close to 0 if the CDDI respects the MaxCPULoad constraint (see formulas (7.21)-(7.23)). Hence, the lower the difference between ACPU(k) and MaxCPULoad, the lower $\Delta PQ_{MAX}(k)$. Then, we raise LoadRatio(k) to the power of 1/CLM to make $\Delta PQ_{MAX}(k)$ higher as CPU load misuse continues. Finally, since $(\text{LoadRatio}(k))^{1/CLM}$ is in [0; 1], we multiply it by $PQ_{MAX}(k)/2$ in order to scale the adaptation action in [0; $PQ_{MAX}(k)/2$], thus keeping it contained with respect to current $PQ_{MAX}(k)$ value.

7.6. Implementation Details

SALES has been implemented and deployed on a real testbed to test the impact of our solutions in real-world systems. In fact, although NS2-based simulations are very useful to study specific data distribution protocols in large-scale mobile networks, they neither model nor consider both CPU and memory overhead introduced on mobile devices. By pursuing our research both on simulations and on real deployments, we aim to reach a more complete view on distribution primitives for context delivery in mobile systems.

With finer details, the real prototype of SALES is completely implemented in Java, and executes on a traditional Java Virtual Machine (JVM) compliant with the 1.6 specifications. In the remainder of this section, we first detail the main components of SALES prototype. Then, we introduce important implementation details: we present the realization of the different transmission policies, and we clarify additional implementation peculiarities used by SALES to limit employed resources on mobile devices. Finally, we introduce the porting of our SALES CDDI on the Android platform, in order to better test the feasibility of our assumptions on real mobile solutions.

7.6.1. SALES Software Architecture

Each node involved into the distributed architecture presented in Figure 7.1 executes a local software instance of the SALES CDDI. Figure 7.5 highlights the main components involved into the CDDI architecture. For the sake of clarity and reusability, SALES exploits two different layers to clearly separate high level context data management issues, from low level data delivery ones. Starting with the upper layer, we now present the main software components included in SALES:

Facility Layer - The facility layer includes all the high-level components useful to handle context data injection and retrieval. From a general viewpoint, it supports context type definition, context data storage, and context data local processing. The *Context Data*

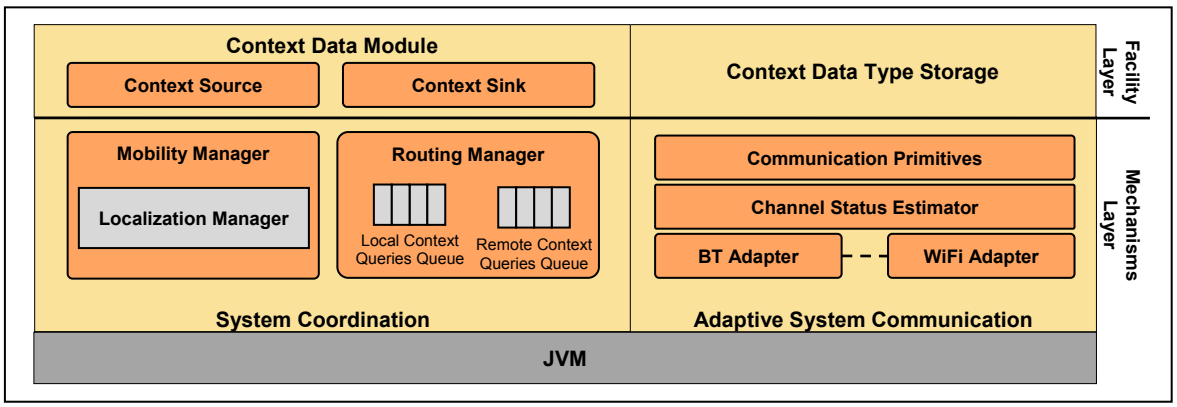

Figure 7.5. SALES Software Architecture.

Type Storage is a local registry useful to store the context data type definitions; each definition is an XML-like file that describes 1) the main attributes contained in each instance of this particular type; and 2) the caching parameters, mainly replacement policy and repository size, used by the different SALES nodes. Each context data type is associated with a *Context Data Module* that enables the real context data injection and retrieval. The general component is then specialized according to each context data type, by also implementing specific routing and caching behaviours as described in the type definition. It also consists of two main sub-modules: the *Context Source* sub-module enables context data generation and injection into the system, while the *Context Sink* allows context data retrieval through context queries.

Mechanism Layer - The mechanism layer implements all the low-level mechanisms useful for inter-nodes communication and hierarchy formation. The adapters, one for each wireless technology, offer technology-independent APIs to the other software components. Currently, SALES includes both a *Bluetooth Adapter* and a *WiFi Adapter* to enable the real communication on the associated technology. On top of them, the *Channel Status Estimator* periodically ascertains the available bandwidth on the different wireless interfaces, while the *Communication Primitives* offers one-way and request/response primitives. The *Adaptive System Communication* wraps the Communication Primitives module to introduce a message-based communication API that, besides hiding wireless technology details, controls and enforces message sending rate; in addition, it offers mechanisms to query the state of the adapters (e.g., available bandwidth, pending messages to send, etc.). By using these APIs, *System Coordination* takes care of handling mobility and context routing. The *Mobility Manager* realizes mobility management protocols to maintain the SALES tree-like architecture; it includes all the main mechanisms to perform the discovery of available father nodes, as well as association procedures. In addition, the *Localization Manager* provides localization information useful to adapt both the mobility management protocols and the context distribution process. Finally, the *Routing Manager* is the main module involved in context data routing: it stores both local and remote context queries, triggers the matching between queries and data, and relays subsequent context responses when required.

7.6.2. Transmission Policies Implementation

In Section 7.5.2, we presented the three transmission policies offered by SALES, namely no batching, naïve batching, and adaptive batching. This section clarifies how the

main components of our CDDI interact at runtime depending on query parameters and adopted transmission techniques, in order to enforce maximum QoC data retrieval time.

Starting from physical message transmission, the different communication adapters take care of sending/receiving messages on the associated wireless technology. To avoid additional overhead, SALES always exchanges messages asynchronously, with no explicit acknowledgment from destination. In addition, each adapter always enforces an outbound data rate below the available bandwidth, that is periodically probed by the Channel Status Estimator (step 1 in Figure 7.6). To avoid strong coupling between receive and local dispatch operations, the System Communication module has a queue, that contains received messages, and a group of dispatching threads (M_D). When a new message is received by an adapter, it is appended into the System Communication queue (step 2), while M_D threads realize the final dispatch to the local manager subscribed for the particular type of message. Here, we focus only on messages used to distribute data/query; they are always dispatched to the Routing Manager (step 3).

Routing Manager receives data instances/queries from local sources and sinks, and queries/responses sent by a remote node from the Adaptive System Communication. Query and response dispatching is implemented by proper query (Q_D) and response (R_D) dispatcher threads that take care of sending query/response messages (step 4-5 in Figure 7.6). Q_D and R_D threads execution is controlled through task descriptors, that are respectively queued in the Query Distribution Task Queue and in the Response Distribution Task Queue. When a new query is received, it is first matched with locally cached data to generate possible responses. If a context query matches a context data, a proper R_D task is queued into the Response Distribution Task Queue (step 6). At the same time, if query parameters require an additional distribution, the query is stored in either the Local Context Queries Queue (if it belongs to a local sink) or the Remote Context Queries

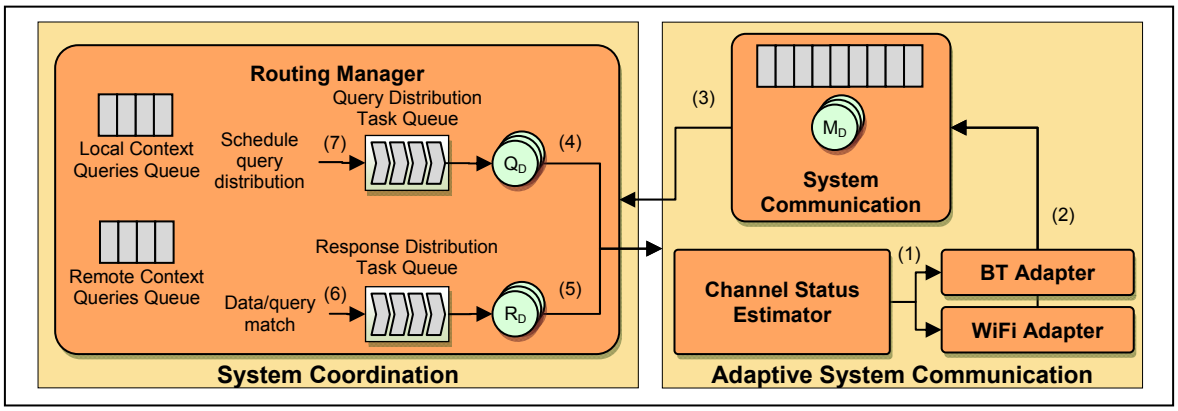

Figure 7.6. SALES Routing Details.

Queue (if it belongs to a remote sink), and a proper Q_D task is scheduled (step 7). Finally, every time a context response is received, the Routing Manager matches contained context data instances with stored queries. If a data instance matches a local query, it is simply inserted into the local context data cache; instead, if a data instance matches a remote query, a new R_D task is scheduled (step 6) to send the data toward the node that had relayed the query. Due to local processing delays, it could be the case that the context distribution task refers to an already expired query; in that case, the task descriptor is silently dropped.

The aforementioned working schema is followed when the no batching transmission policy is adopted. R_D and Q_D executions are automatically triggered by task descriptors expiration, scheduled after a delay equal to DRD/ QRD according to query parameters.

When one of the two batching policies is enabled, R_D and Q_D threads execute a different set of operations toward the goal of batching more data instances/queries in the same message. First of all, the Routing Manager has a local map, indexed by destination node and called *InformationToDistribute* in the remainder, on which data instances/queries to be sent are queued. Every time the Routing Manager schedules a data instance/query distribution, it actually creates two task descriptors in the appropriate queue: one is associated with the lower bound of the distribution period, while the other marks the upper bound. Hence, the first one, executed after a delay of DRD/QRD, signals the need of distributing a data instance/query to a particular destination; main goal of this task is to queue the associated data instance/query in the local InformationToDistribute map, in order to keep track of the data/queries that can be distributed. Instead, the second one, executed after a delay of either DRD_M/QRD_M (in case of naïve batching) or ADRD/AQRD (in case of adaptive batching), triggers the real transmission; every time a task descriptor of such kind expires, the associated thread collects all the data instances/queries for the same destination from the InformationToDistribute map, and batches them in a unique message. Of course, if a data instance/query is distributed before the expiration of its own second task descriptor due to batching, the Routing Manager automatically cancels that task descriptor since not useful anymore.

Hence, when we enable batching techniques, we use the InformationToDistribute map as synchronization means to understand which data instances/queries have to be distributed. By using this solution, on the one side, SALES does not need to iterate over all the current descriptors, thus experiencing reduced CPU load. On the other side, this solution increases the memory overhead due to the additional InformationToDistribute

map. However, since current mobile devices usually have CPU limitations tighter than memory ones, we decided to adopt such solution.

7.6.3. Resource-aware Components

To limit employed resources, SALES provides a set of configuration parameters that specify both processing and memory resources available to components showed in Figure 7.6. Processing resource configuration parameters mainly include the number of threads for each thread pool and the maximum number of requests processed per second. Instead, memory resource configuration parameters mainly concern maximum queue lengths. For the sake of clarity, since all the queues in Figure 7.6 can contain either thread task descriptors or messages, in the remainder we use the generic term "element" for queue management policies description.

Starting with processing resources, as presented in Section 7.5.3, SALES adopts an adaptive threshold to limit the maximum CPU load associated with query processing. Toward this direction, the Routing Manager component proactively discharges query received by neighbours. However, the usage of such a dropping policy does not allow the fine control of the management overhead introduced by threads contained in both the Adaptive System Communication and in the single Communications Adapters. In fact, even if the query is locally discarded by the Routing Manager, the Adaptive System Communication always introduces an additional overhead consequence of message receive and decoding operations. Hence, SALES associates each thread pool with a maximum number of threads and a maximum execution rate for each one of them. When the number of received messages is higher than the total processing rate of the Adaptive System Communication (we remark that this component also handles messages related to mobility management), pending requests will experience queuing delays that, for instance, increase the probability of not respecting QoC data retrieval time. To maximize satisfied requests, SALES reactively recovers queuing delays by dynamically changing the DRD/QRD parameters carried by a context query: every time a Q_D thread processes a query whose initial message suffered of queuing delays in the Adaptive System Communication, it considers the number of hops contained in the worst-case distribution scenario, and reduces DRD/QRD by considering the ratio between the delay and the total number of hops, so to recover the time unnecessary spent in the queue.

Moving to memory management, SALES limits the length of all the data structures involved in Figure 7.6. All the queues involved in both the Adaptive System

Communication and in the Communication Adapters have a maximum length, statically imposed by SALES configuration file; when a queue is full, different policies can be adopted to select the element to remove. At default, SALES applies a traditional First In-First Out (FIFO) policy; apart from that, the queues containing the task descriptors into the Routing Manager can also apply priority-based policies, so as to favour the routing of context data associated with high priority clients. Similar limitations have to be imposed to the context query tables contained into the Routing Manager. In fact, the storage of context queries not only introduces increased memory overhead, but also leads to additional context data distributions, thus finally increasing CPU and bandwidth overhead. Finally, as also presented in Section 7.5.1, each Context Data Module has a local data repository useful to store important context data, whose maximum size D_{MAX} is limited according to available memory.

7.6.4. SALES on the Android platform

To better assess the feasibility of our SALES CDDI in real-world systems, we decided to port it on Android since one of the most widespread platforms for mobile devices. In this section, for the sake of completeness, we first introduce a brief overview of the Android platform; then, we present how the main limitations introduced by this platform impact on the realization of SALES distribution primitives.

Android is a software stack for mobile devices which includes an operating system, a middleware layer, and a set of support services. The Android Software Development Kit (SDK) is based on Java, and offers all the tools necessary to develop applications on such platform. The software stack is based on a Linux kernel and offers usual system services, such as process and memory management, networking support, etc. On top of the Linux kernel, we find both the Android runtime, that provides most of the functionalities available in Java, and a set of C/C++ utility libraries, that implement efficient media libraries for codecs, SQLite for relational databases, and so forth. Each Android application executes in its own process, with its own instance of the Dalvik Virtual Machine (DVM).

In finer details, the *Application Framework* includes the main Android components: a set of *Views* to manage the user interface, *Content Providers* to store and share data between applications, a *Resource Manager* to provide access to non-code resources (e.g., strings, images, etc.), a *Preference Manager* to store configuration parameters of applications, and an *Activity Manager* to handle application lifecycle. Android introduces

an application model that clearly separates presentation logic from data storage and background services; in particular, it distinguishes four main component types [119]:

- *Activities* - An activity represents a single screen user interface. Each application can be composed by several activities, independent from each other. Activity lifecycle is handled by the Activity Manager: an activity is started when it becomes visible, paused or stopped if another activity becomes visible, and destroyed if no longer used.
- *Services* - A service is a background component that performs long-running operations. Differently from activities, services do not directly interact with users; instead, they receive commands from either activities or other services. While the lifecycle of an activity is strictly related with its own visibility on device screen, services are supposed to execute without interruption. In general, a service is killed by the Application Framework only in critical conditions, such as device running out of memory.
- *Content Providers* - A content provider is in charge of managing a shared set of application data irrespective of their location. Data can be located on the local file system, on a SQLite database, or even on the web. Content providers can be queried by applications running on the local device through proper SQL-like commands; similarly, new data can be dynamically added and removed, while having a common storage available for all running applications.
- *Broadcast Receivers* - A broadcast receiver is a component that responds to system-wide announcements, originated either from the system or from other applications. In Android, broadcast receivers are fundamental to implement asynchronous and anonymous communication mechanisms, where senders and receives do not have to know each other. Such mechanisms are usually used to dynamically start and stop services, in order to reduce mobile device overhead.

Apart from those main components, the DVM includes all the main libraries and classes available on a traditional JVM. However, the two VMs are not completely aligned: on the one side, DVM does not realize some libraries available on standard JVM, such as advanced data structures for concurrent programming; on the other side, DVM adds new packages that do not adhere to some accepted standards in the Java world, such as for Bluetooth, where Android does not follow the JSR-82 proposal [120], but introduces its own new *android.bluetooth.** package.

In view of the Android application model, we designed our SALES CDDI client, by clearly separating user interactions from background services and context data storage. In particular, we introduced three main components, namely *CDDI Context Data Provider*, *CDDI Service*, and *CDDI Configuration Activity*, better detailed in the following. For the sake of clarity, Figure 7.7 shows the logical architecture of the CDDI client, by also remarking how the main SALES components are mapped in the Android application model.

CDDI Context Data Provider interacts with local context-aware applications to enable context data injection and retrieval, thus enacting as a bridge between CDDI and running applications. All the local context data type definitions are available to external applications through a standard URL. In addition, each Context Data Module is wrapped in a Content Provider that takes care of saving data instances in a local SQLite database, and relays not satisfied queries to the CDDI Service, so to point out local context needs that are currently not satisfied.

CDDI Service contains all the low-level mechanisms involved in both maintenance of the system distributed architecture and in context data distribution process. Apart from the main components already presented in Section 7.6.1, here, we also introduce a Wireless Card Dynamic Configuration module in charge of dynamically reconfiguring available network interfaces according to the current execution context. In fact, in real-world scenarios, SALES has to explicitly deal with wireless network interfaces, by dynamically changing configuration parameters to reach high-level goals: for instance, since SUNs do

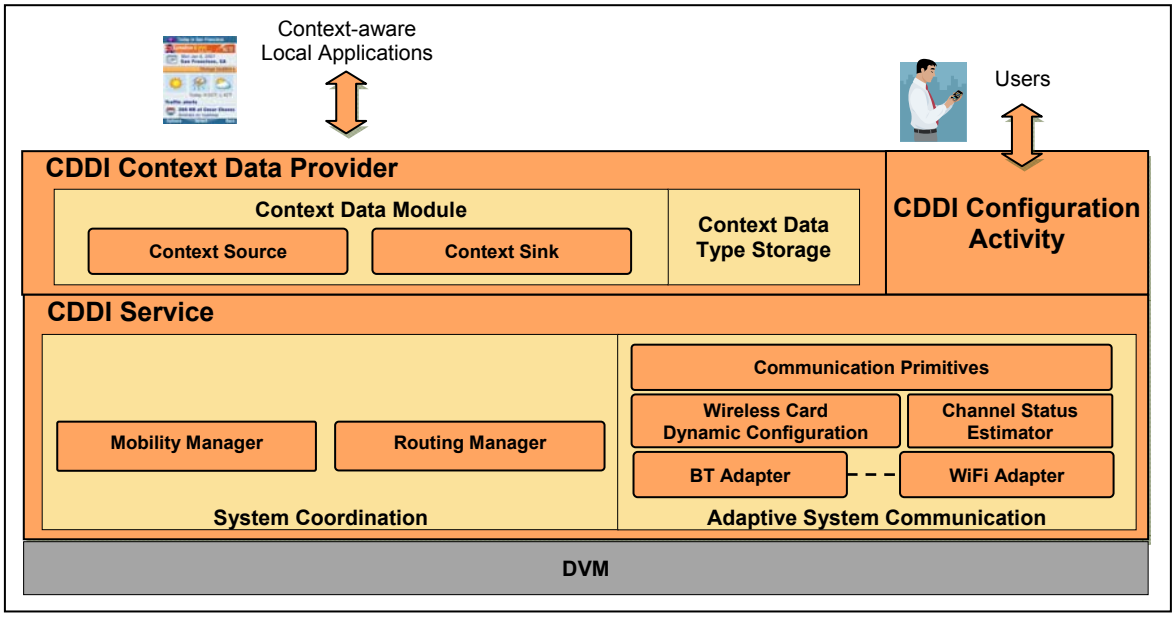

Figure 7.7. SALES Android-based Client.

not need a connection to the wireless fixed infrastructure, it can dynamically switch off network interfaces to save battery lifetime.

Finally, CDDI Configuration Activity allows users to modify configuration parameters. To store and modify such information, this component is built upon the default Android Preference Manager, that permits to define configuration views via simple XML documents. As specified in the previous section, SALES lets users to limit the amount of allocated resources, such as CPU, memory, and bandwidth, to avoid an intolerable overhead.

Moving to finer implementation details, Android imposes important constraints in two main areas: wireless networks configuration and memory management. Before proceeding further, we remark that all the following considerations assume an unmodified Android version 2.2. We omitted complex workarounds to these problems since we are interested in evaluating the feasibility of CDDIs based on a standard Android distribution, rather than in introducing complex solutions difficult to be deployed, maintained and used by normal users.

Starting from wireless networks configuration, Android does not allow the usage of WiFi cards in ad-hoc mode. Hence, an Android phone can only connect to WiFi infrastructure-based networks, thus hindering the context distribution process on SALES mobile infrastructure. A partial workaround is available if we use the Android WiFi tethering facility; in this mode, a mobile device enacts as an AP, while others connect to it as they would have done with a real AP. Unfortunately, this solution has an important shortcoming: since devices do not form an ad-hoc network, all the transmissions have to pass through the node acting as the AP that, in its turn, relays them to the real destination. Of course, that reduces network performance.

Focusing on BT connections, Android uses the *android.bluetooth.** package that exploits Broadcast Receivers to signal BT-related events, e.g., start and stop of the discovery process, new mobile devices discovered, and so forth. However, provided APIs are extremely limited, and do not enable fine and direct control of both discovery and connection process. Above all, Android requires direct user confirmation to set the device in discoverable mode, even if the discovery phase has been triggered by the CDDI Service. In addition, for the sake of battery lifetime, Android limits the length of discovery time to no longer than 300 seconds; after that, it is necessary to prompt again the user to make the device discoverable again. Moreover, during the first connection between two devices, Android requires a manual pairing process that also needs explicit user

intervention. Finally, Android APIs offer only connection-oriented Radio Frequency COMMunication (RFCOMM) links that, by construction, introduce higher management overhead in respect of simple Logical Link Control and Adaptation Protocol (L2CAP) links [121]. Hence, all these constraints, even if perfectly reasonable when battery preservation and user privacy are the main goals, widely limit opportunistic exploitation and transparent management of ad-hoc BT links.

By considering all these limitations together, the Wireless Card Dynamic Configuration has very limited degrees of freedom on a traditional Android version 2.2 installation. Android APIs do not allow dynamic configuration of the wireless network interfaces by Java code, and the user has to be explicitly involved into the wireless connection configuration and setup process for both WiFi and BT. These constraints really clash with the main requirements of dynamically and transparently reconfiguring the different wireless cards at the SALES CDDI client. We believe a limited control on wireless interfaces, for instance via a standard Android manager, would greatly ease the realization of real-world network middleware supports with minimal user intervention.

Moving to memory management, there is a fundamental difference between Dalvik Garbage Collector (GC) and traditional Java GC mechanisms. Above all, to preserve battery lifetime and reduce computational overhead, Dalvik GC applies lazy collection policies and does not perform dynamic heap memory relocation. Hence, especially when Java objects have variable sizes, the Dalvik heap can suffer of high fragmentation, thus perhaps leading to high heap space waste. Apart from a careful reuse of Java objects, if the application uses large byte arrays, for instance due to data serialization, the programmer should introduce additional mechanisms in charge of splitting them in smaller and fixed-sized chunks; in this way, subsequent memory allocations can be satisfied by using pre-existing heap chunks freed in the meantime, thus not adding to heap fragmentation.

To conclude, at the current stage, the deployment of real-world CDDIs for the Android platform introduces particular issues that have to be carefully handled. Similarly to SALES, many research works assume to dynamically reconfigure wireless network interfaces, by also using WiFi ad-hoc links to create MANETs for service delivery. All these assumptions do not fit well the Android mobile platform, which imposes tight constraints on wireless network cards reconfigurations from Java code. At the same time, CPU and memory limitations of traditional mobile devices can require a better tailoring of the main solutions introduced by SALES. In Section 7.7.4, we present experimental results about our real Android-based client, so as to better remark possible performance

limitations due to resource-constrained mobile devices.

7.7. Experimental Results

SALES has been implemented and deployed in 1) NS2 simulations, in order to validate our protocols in large-scale mobile systems; and 2) a real wireless testbed, in order to test the feasibility of the main mechanisms introduced in this chapter on real mobile devices. Although our work on SALES mainly focused on real-world deployments, in Section 7.7.1 we exploit simulations to test our ACDC caching protocol; we opted for this choice since NS2 simulations let us to better evaluate the technical soundness of our proposal in large-scale systems, where several mobile devices share context data among themselves while roaming. Instead, in Section 7.7.2, Section 7.7.3, and Section 7.7.4, we consider the real-world implementation of SALES, so to better highlight system management overhead and the real feasibility of our proposals on real mobile devices. Let us now anticipate important details about NS2 simulation parameters and real-world implementation.

Starting with NS2 simulations, if not stated differently, we consider a simulation area of 350x350m with 50 nodes, randomly roaming according to RWP model (uniform speed in [1; 2] meters/second and a uniform distributed pause in [0; 10] seconds). Each node has two wireless interfaces, both based on IEEE 802.11g technology (bandwidth of 54 Mbps) and with a transmission range of 100m. Each node emits a mobility beacon with a period of 10 seconds to signal its presence, and dynamically discovers and associates with available BNs. The simulation area is covered by 5 APs, each one connected to a different BN, respectively placed in [175; 175], [100; 100], [250; 250], [250; 100], and [100; 250]; due to adopted transmission ranges, the area is almost entirely covered by fixed connectivity. Finally, all simulations last 15 minutes (900 seconds), and reported results are average values over 33 runs with different RWP instances. Additional details about context data production and retrieval will be clarified in Section 7.7.1.

Moving to the real implementation (used in Section 7.7.2, Section 7.7.3, and Section 7.7.4), SALES fixed infrastructure is composed by one CN and two BNs, all of them running on Linux-based boxes with 3GHz CPU and 2GB RAM. The BNs offer infrastructure-based connectivity to mobile devices by means of traditional IEEE 802.11g Cisco APs. Instead, as regards the mobile infrastructure, we have used a mix of laptops and mobile phones, arranged with different configurations clarified in each one of the following tests. Each laptop has an Intel Core Duo 2 T6500 and 4 GB RAM, while each

mobile phone is an LG-P500, based on Android version 2.2 and equipped with both a WiFi and a BT interface. Moving to the software architecture, SALES is fully implemented in Java. Hence, it needs either a traditional JVM 1.6 when executed on laptops, or a Dalvik VM when deployed on Android phones. As stated before, the two implementations present some significant differences due to the unavailability of standard Java 1.6 classes on Android 2.2 platform.

In the remainder, we present experimental results about the main mechanisms introduced in this chapter. We start with NS2 simulations to validate ACDC data caching; then, we use the SALES real implementation to test both data/query transmission techniques and query dropping policies. Finally, we present novel results to compare key performance metrics according to whether we use SALES CUN/SUN J2SE-based implementation on full-fledged laptops or Android-based implementation on resource-constrained mobile phones.

7.7.1. ACDC Data Caching Evaluation

In SALES, context data caching is fundamental to enable efficient and effective wireless infrastructure offloading. Mobile devices share cached data with neighbours by ad-hoc links, thus perhaps reducing the final traffic to/from the wireless fixed infrastructure. For the sake of technical evaluation, the NS2 implementation of SALES considers only BNs and CUNs; that allows us to better evaluate infrastructure offloading capabilities, while leaving out possible side-effects introduced by mobile nodes clustering.

Focusing on context data production and retrieval, we consider 1000 sources, all deployed on the fixed infrastructure and equally divided among BNs. Each data instance has a payload of 3KB, so as to simulate worst-case scenarios where context data contain images or serialized user/place profiles. Each context source periodically produces a new data with a FL parameter (see Section 7.3) equal to the generation period; if not stated otherwise, both generation periods and data FLs are equal to 180 seconds, in order to test the more challenging case of short lived data. Each CUN can cache a maximum number of context data instances equal to 30. If needed, data replacement is carried out through one of the following policies: LRU, LFU, ACDC_OL, and ACDC. While LRU and LFU are the traditional replacement policies as clarified in Section 7.3.1, ACDC is our novel proposal presented in Section 7.3.2. In addition, for the sake of completeness, we also consider a simplified version of ACDC, which exploits “Only Local” (OL) rank, to better understand the effects of local and remote ranking in our full ACDC proposal.

As regards context query production, each CUN periodically emits a new context query directed to a specific source, that is selected by one of the following two policies. The first one follows a uniform distribution: the CUN randomly selects the source between [0; 999], hence, all sources have the same probability of being accessed. The second one is a localization-based preferential distribution: we superimpose a 10x10 virtual grid over the simulation area and, for each cell, called *virtual cell* in the remainder, we use a different Gaussian distribution to choose the final source to query; the average of the Gaussian distribution depends on the cell in which the node is currently in, and neighbouring cells have overlapping distributions to mimic localization-based accesses. We used these two distributions since the first one mimics a worst-case scenario where data caching on the mobile infrastructure is not effectively exploited, while the second one models a wide set of realistic scenarios where CUNs in physical proximity share common interests and access the same sources.

Finally, let us clarify the main performance indicators we considered. First, we compare the average retrieval time experienced by a CUN to access the context data instance belonging to the requested source, namely the time between query emission and data delivery to sender node. Second, we consider the percentage of satisfied queries, so to better stress the impact on the reliability of the distribution process. Finally, to evaluate infrastructure offloading, we consider three traffic indicators, namely 1) the cumulative traffic sent from the fixed to the mobile infrastructure ($T_{IF \rightarrow MF}$); 2) the cumulative traffic sent from the mobile to the fixed infrastructure ($T_{MF \rightarrow IF}$); and 3) the cumulative traffic sent on ad-hoc links ($T_{AD\text{-}HOC}$).

In the first set of experiments, we start by comparing ACDC and the other caching policies with uniform access patterns and different HTTL values. Figure 7.8 (a) and Figure 7.8 (b) show respectively the average retrieval time and the percentage of satisfied requests with caching policies in {LRU, LFU, ACDC_OL, ACDC} and HTTL in {1, 2,

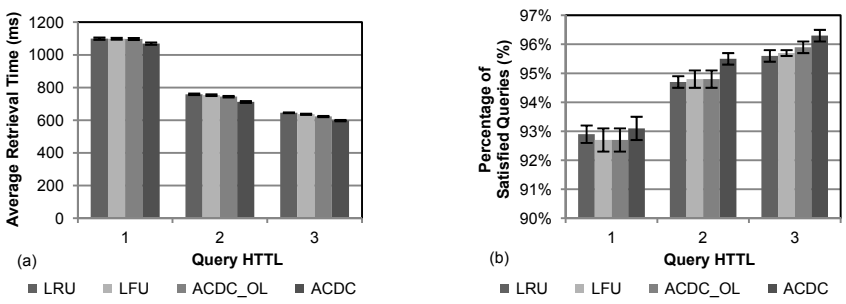

Figure 7.8. Average Retrieval Time (a) and Percentage of Satisfied Queries (b) under Uniform Access Patterns.

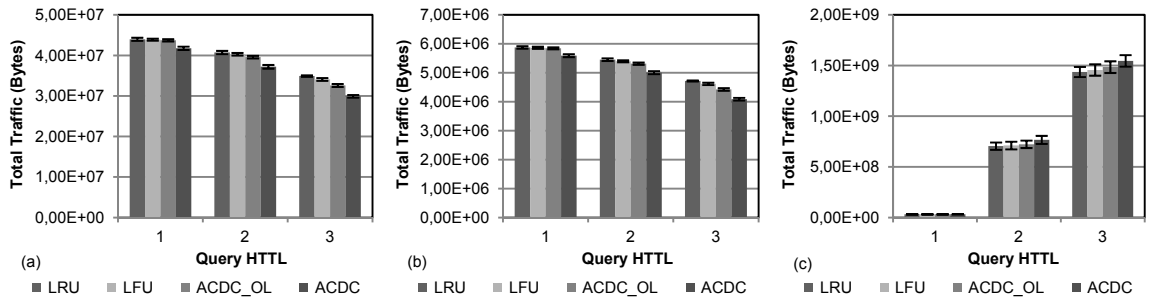

Figure 7.9. $T_{IF \rightarrow MF}$ (a), $T_{MF \rightarrow IF}$ (b), and $T_{AD\text{-}HOC}$ (c) according to Different Caching Algorithms and Query HTTL, under Uniform Access Patterns.

3}. At the same time, for the sake of clarity, Figure 7.9 (a), Figure 7.9 (b), and Figure 7.9 (c) show the cumulative $T_{IF \rightarrow MF}$, $T_{MF \rightarrow IF}$, and $T_{AD\text{-}HOC}$ for considered test configurations. We have to note that, in all experiments, our ACDC approach reduces the traffic to/from the fixed infrastructure due to the remote ranking that, in its turn, leads to increased data diversity between repositories in physical proximity. However, it is important to note that the reliability of the distribution process is only slightly improved (see Figure 7.8 (b)): that is due to the fact that, although routing fails on the mobile infrastructure, the CUN can retrieve needed data from the fixed infrastructure, thus increasing the total $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$. Finally, it is interesting to compare ACDC_OL and ACDC. The latter always outperforms the former due to higher repository diversity, thus leading to lower $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$; unfortunately, at the same time, ACDC leads to increased $T_{AD\text{-}HOC}$ since the higher data repository diversity also increases the probability that each query reaches a wider set of context data.

In the second set of experiments, we considered more realistic localization-based preferential access patterns; the Gaussian distribution exploited in each cell has a Standard Deviation (S.D.) of 26, so as to prevent that most of the queries find a positive response directly from the local cache deployed at the sender node. Here, we expect better performance since CUNs in physical proximity require the same set of data, thus leading

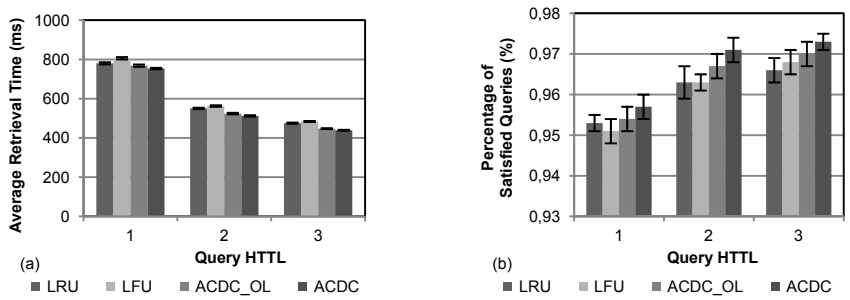

Figure 7.10. Average Retrieval Time (a) and Percentage of Satisfied Queries (b) under Localization-based Preferential Access Patterns.

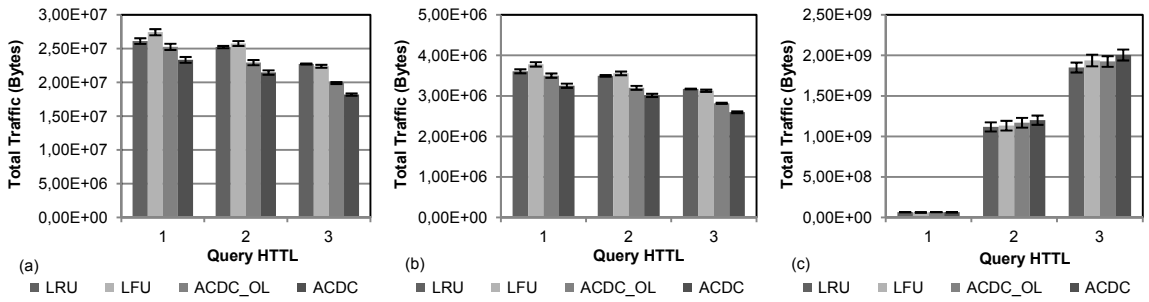

Figure 7.11. $T_{IF \rightarrow MF}$ (a), $T_{MF \rightarrow IF}$ (b), and $T_{AD\text{-}HOC}$ (c) according to Different Caching Algorithms and Query HTTL, under Localization-based Preferential Access Patterns.

to better infrastructure offloading. Similarly to previous experiments, Figure 7.10 represents average retrieval times and percentage of satisfied requests, while Figure 7.11 shows cumulative $T_{IF \rightarrow MF}$, $T_{MF \rightarrow IF}$, and $T_{AD\text{-}HOC}$ at the end of the simulation. In respect of uniform access patterns (see Figure 7.8 (a)), here we experience lower average retrieval times and higher reliability due to the higher similarity of emitted context queries. First of all, it is interesting to note that LFU leads to the worst performance since that caching approach tends to integrate the whole history of accesses, hence, it does not adapt well when access patterns change due to CUNs roaming between different virtual cells of the simulation area. Also, similarly to what we found in previous experiments, ACDC is the best caching solution between considered ones, while ACDC_OL is the second best one. Focusing on Figure 7.11, we remark that, in respect of Figure 7.9, both $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$ are smaller, thus further increasing infrastructure offloading. Unfortunately, $T_{AD\text{-}HOC}$ increases as a higher number of close CUNs cache matching data, thus triggering a higher number of responses.

From above results, we conclude that both ACDC_OL and ACDC outperform other caching approaches. In both uniform and localization-based preferential access patterns, they increase infrastructure offloading; in addition, ACDC usually performs better due to increased data repository diversity, but also leads to higher traffic on ad-hoc links due to the increased number of triggered responses. In all the previous experiments, we exploited a fixed data FL of 180 seconds and a query generator S.D. of 26; now, we want to evaluate the effects of such parameters on infrastructure offloading. Let us also remark that, for the sake of conciseness, in the remainder we only consider localization-based preferential access patterns as they are more realistic and allow real offloading through caching.

In the third set of experiments, we considered data with longer FLs to test the performance of the different caching approaches with long lived context data. In fact, short lived data can either hinder or help context data caching: on the one side, since data are

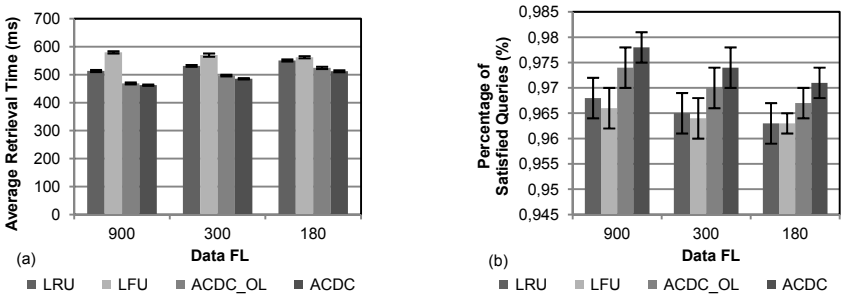

Figure 7.12. Effect of Different Data RL Values on Average Retrieval Time (a) and Percentage of Satisfied Queries (b).

automatically removed due to RL expiration, we periodically need to pull again the data from the BNs; on the other hand, especially for those approaches, e.g., LFU, that keep track of data accesses through history mechanisms, data removal due to RL expiration could be beneficial as it allows to flush context data and associated history, thus allowing faster context data cache adaptations. Figure 7.12 shows the average retrieval times and the percentage of satisfied queries with data RL in {900, 300, 180} seconds, while Figure 7.13 shows the cumulative $T_{IF \rightarrow MF}$, $T_{MF \rightarrow IF}$, and $T_{AD\text{-}HOC}$ for the current test configuration. Starting with Figure 7.12, we note that LFU ensures the worst performance, especially for long lived data; again, this is due to the fact that LFU accumulates all the access history, thus hindering the fast adaption of caches. We remark that, if data FL is 900 seconds, context data never expire during the simulation, and are removed only for data replacement due to memory saturation. By analyzing Figure 7.13, we note that ACDC_OL and ACDC are always the ones that ensure lower $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$, thus further increasing infrastructure offloading. Of course, the higher the data FL value, the lower the traffic with the infrastructure will be, since context data will be probably kept alive on CUNs and fetched from them. Also here, we note that LFU history effects lead to higher traffic with the fixed infrastructure.

In the fourth set of experiments, we consider S.D. values in {13, 26, 52, 104} for the Gaussian distribution used to select the interesting source in each virtual cell. Of course,

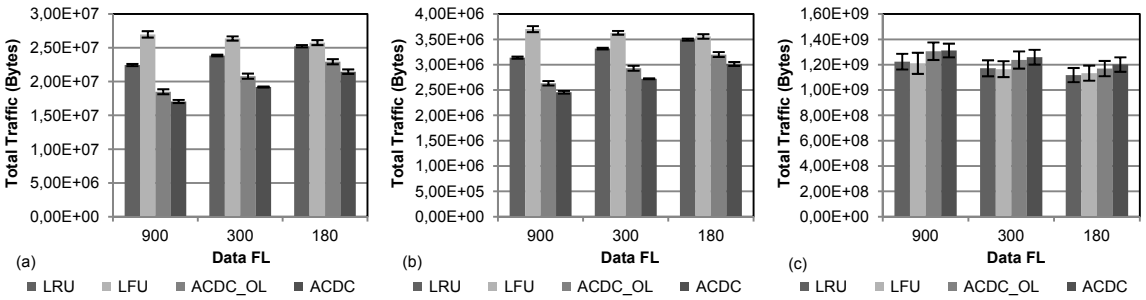

Figure 7.13. $T_{IF \rightarrow MF}$ (a), $T_{MF \rightarrow IF}$ (b), and $T_{AD\text{-}HOC}$ (c) with Different Data RL.

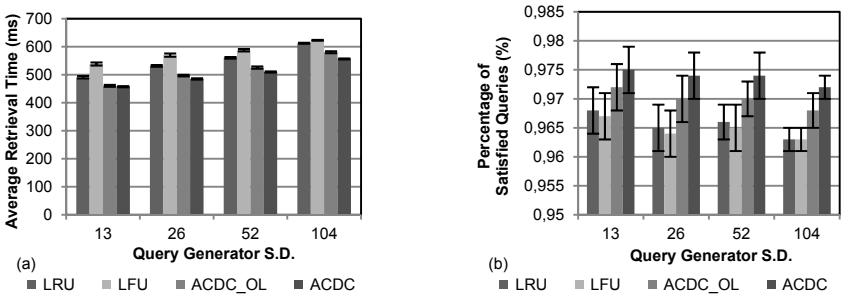

Figure 7.14. Effect of Different Query Generator S.D. Values on Average Retrieval Time (a) and Percentage of Satisfied Queries (b).

higher S.D. values reduce caching usefulness since each cell will be associated with a wider set of interesting context data sources. Figure 7.14 and Figure 7.15 present the same performance indicators used in previous tests. Similarly to what happened before, LFU is the worst caching algorithm as it leads to higher retrieval times and lower percentage of satisfied requests. With higher S.D. values, average retrieval times tend to increase as context data will be probably cached in farther nodes (see Figure 7.14 (a)). With an S.D. value of 104, LRU and LFU perform very similarly since LFU suffers reduced history effects. However, in all the considered configuration tests, our proposals, namely both ACDC_OL and ACDC, are the better ones. From Figure 7.15, we confirm that our two proposals lead to reduced traffic to/from the infrastructure, thus improving the final offloading. Also, ACDC always performs better than ACDC_OL in terms of $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$, although it leads to slightly higher $T_{AD\text{-}HOC}$ traffic due to increased data repository diversity. Finally, in general, we remark that higher S.D. values lead to 1) increased $T_{IF \rightarrow MF}$ and $T_{MF \rightarrow IF}$ since more context data instances need to be fetched from the fixed infrastructure; and 2) reduced $T_{AD\text{-}HOC}$ since each query will trigger a reduced number of context responses due to the larger set of context data stored on CUNs in physical proximity.

Hence, we conclude that, in all the considered test configurations, both ACDC_OL and ACDC continue to outperform LRU and LFU. In addition, ACDC usually performs better than ACDC_OL in terms of infrastructure offloading, since it is able to increase data repository diversity between close CUNs. Unfortunately, it also increases traffic on ad-hoc links since each query can trigger a higher number of responses. However, since our main objective is to improve infrastructure offloading for the sake of scalability, and considering that ad-hoc links do not usually introduce economical costs for the infrastructure provider, we claim that ACDC is a feasible solution to efficiently and effectively offload the wireless fixed infrastructure.

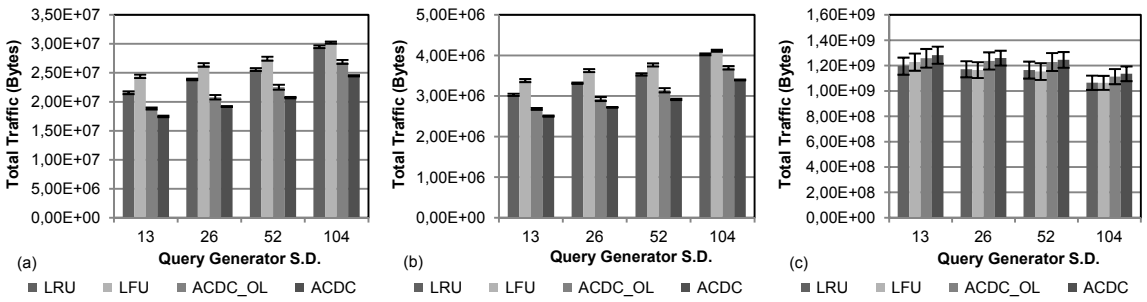

Figure 7.15. $T_{IF \rightarrow MF}$ (a), $T_{MF \rightarrow IF}$ (b), and $T_{AD\text{-}HOC}$ (c) with Different Query Generator S.D.

7.7.2. Data/query Transmission Policies Evaluation

In this section, we present experimental results about the different transmission policies, e.g., no batching, naïve batching, and adaptive batching, offered by SALES CDDI. As stated before, from now on, we consider our real-world implementation of SALES to test the feasibility of such mechanisms and the introduced management overhead on real mobile devices. Considering that obtained results truly depend on adopted communication primitives, let us remark that SALES adopts UDP as transport protocol. Messages do not receive explicit acknowledgements from destination, hence, message droppings due to packet collision, socket buffer overflow, and so forth, are possible.

In addition, we configured SALES system-level resources as follows (see Section 7.6.3). To emulate SUNs/CUNs over cellular phones or PDAs, each CUN/SUN has 3 M_D, 1 R_D, and 1 Q_D threads, each one executing (if not explicitly stated) at most 50 reqs/s. Instead, both the CN and the BNs have 10 M_D, 3 R_D, and 3 Q_D threads, each one executing at most 60 reqs/s. All the involved queues have a maximum size of 100 elements on CUNs/SUNs, and of 200 elements on CN/BNs. The mapping between the maximum QoC data retrieval time and the query parameters exploits an α parameter of 0.8 and a γ parameter of 2. An average bandwidth of 6 Mbps is available on ad-hoc links, while wireless network load factors are exchanged every 10 seconds. Finally, as we did before, all the showed experimental results are average over 33 test executions to obtain a good confidence; standard deviation is also presented.

Focusing on context data production and consumption, similarly to previous experiments, we exploit a context data type with a payload size of about 3 KB. To simulate the worst-case scenario, namely the longest possible distribution path, we have deployed 1000 context sources on the CN, and each source continuously produces data instances with a FL parameter uniformly distributed in [150; 300] seconds. For the sake of

repeatability, and since we are now interested in evaluating only the effect of the different transmission policies on wireless links, BNs, CUNs, and SUNs do not cache any data: hence, all the queries have to reach the CN to find a positive response. Finally, in the following, each emitted query is directed to a particular context source, randomly selected at the query creator node with a uniform distribution. We remark that this choice has been made to avoid overlapping queries that, in their turns, could reduce the total number of data transmissions; by doing in this way, we can manually estimate the expected network traffic load.

In the first set of experiments (see Figure 7.16), we have compared the three data/query transmission policies presented in Section 7.5.2. Toward this goal, and to neatly separate the effect of our policies, all the nodes involved in this set of experiments have 1) a local processing rate, namely M_D, R_D, and Q_D execution rate, equal to twice the request rate; 2) the query dropping policy disabled; and 3) the reactive routing delays adaptation disabled. To consider very challenging scenarios, we decided to adopt request rates in {50, 55, 60, 65, 70} reqs/s, and a data retrieval time of 2 seconds. Although such request rates could seem very high and unrealistic, we remark that they are reasonable if we consider that our SUN actually simulates a set of mobile nodes attached to the same CUN. In densely populated environments, such as a university classroom, we can find hundreds of mobile devices in the same physical place, whose wireless transmissions always interfere among them; hence, proposed workloads are feasible since, from the wireless network viewpoint, they actually mimic scenarios where each mobile device emits less than 1 reqs/s. Finally, we imposed 1) query HTTL equal to 0 (no query horizontal distribution is performed); and 2) access to the latest version of the context data (each query has to reach the context source on the CN). The SUN_{111} executes our test code, and requests a fixed number of reqs/s for a long test of 5 minutes.

Depending on the adopted transmission technique, Figure 7.16 (a) and Figure 7.16 (b)

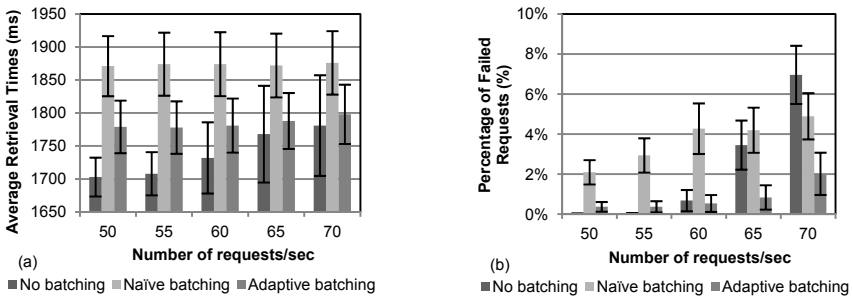

Figure 7.16. Average Retrieval Time (a) and Percentage of Failed Requests (b) with Different Transmission Policies.

show respectively the average retrieval time, namely the time interval between query generation and response arrival, and the percentage of failed requests for different request rates. Except for request rate of 70 reqs/s, as regards context retrieval times, we remark that 1) no batching policy outperforms both naïve and adaptive batching since they use larger delays to trigger real transmissions; and 2) adaptive batching outperforms naïve batching due to the inherently lower transmission delays, obtained by means of the θ parameter. Instead, with a request rate of 70 reqs/s, the adaptive batching performs very close to the no batching. In fact, the increased wireless congestion of the no batching policy frequently triggers MAC backoff mechanism that, in its turn, results in increased delays during wireless channel access, hence higher total retrieval times. Finally, while the adaptive batching features quite similar retrieval times with different request rates, the no batching is very sensible to this parameter due to the increased wireless congestion.

Figure 7.16 (b) shows the percentage of failed requests. We consider a failure either a request without a response (hence, consequence of a message drop) or a request with a late response (hence, received after the data retrieval time). No batching is the best choice when request rate is in {50, 55} reqs/s: the lower transmission delays can balance unforeseen delays, thus leading to a lower number of late responses. Instead, from 60 reqs/s, the adaptive batching outperforms no batching. In fact, no batching policy increases wireless channel congestion: this, in its turn, leads to increased late responses and message droppings. In addition, to better clarify obtained results, we used a background process to periodically ping the CUN from the SUN; we found out that, with a request rate of 70 reqs/s, ping times reach more than 300 ms, and that explains the sharp increase of late responses. Finally, except for request rate of 70 reqs/s, the naïve batching is always the worst policy as higher routing delays can likely lead to failures due to late responses.

From above results, we remark that 1) the no batching ensures the lowest retrieval times and the highest reliability for low request rates; 2) the naïve batching outperforms no batching in reliability only for very high request rates, but it usually leads to a high number of failures due to late responses; and 3) adaptive batching performs very close to the no batching for low request rates, and outperforms it with high request rates. In particular, adaptive batching has both self-optimization (it automatically finds a tradeoff between timeliness and reliability) and self-configuration (it automatically reconfigures all the required SALES components) capabilities in respect of wireless channel congestion.

In the second set of experiments, we used the above configuration test to analyze the effect of the α parameter, used to evaluate distribution period lower bounds, on the three

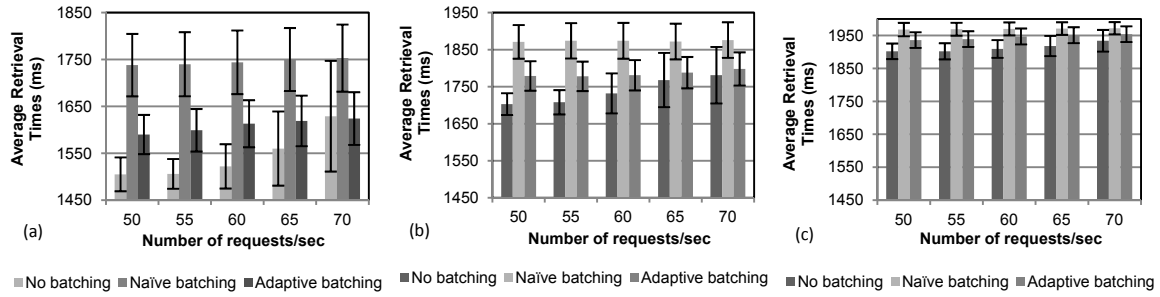

Figure 7.17. Average Retrieval Times with α in {0.7 (a), 0.8 (b), 0.9 (c)}.

transmission policies. By using an α in {0.7, 0.8, 0.9}, Figure 7.17 and Figure 7.18 show respectively the average retrieval times and the percentage of failed requests for different request rates. First, by comparing Figure 7.17 (a), Figure 7.17 (b), and Figure 7.17 (c), we note that the lower the α parameter, the lower the retrieval times; in fact, lower α values lead to lower DRD and QRD, thus possibly anticipating data/query transmissions. In addition, by comparing Figure 7.18 (a), Figure 7.18 (b), and Figure 7.18 (c), we remark that the lower the α parameter, the lower the percentage of failed requests; in fact, lower α values anticipate query/data transmission, thus giving more chances to recover unforeseen delays introduced by system congestion. When batching is enabled, lower α values increase also the probability of data/query batching, thus further reducing wireless channel accesses and network congestion. From these results, we conclude that α parameter is useful to trade off reliability and data retrieval time. On the one side, lower α values are appealing since able to reduce the percentage of failed responses. Unfortunately, on the other side, lower α values anticipate context data distribution, thus hindering the usage of routing delays at each single node to prevent system congestion.

In the last set of experiments, we focused on the adaptive batching technique to test its behaviour under time-varying workloads. By limiting the processing rate of mobile nodes to 50 reqs/s, we execute a test of 35 minutes divided in 7 different timeslots: each timeslot is 5 minutes (300 seconds) long, and employs a static request rate to stress SALES. The adopted request rates are respectively (10, 30, 50, 70, 50, 30, 10) reqs/s. Figure 7.19 shows

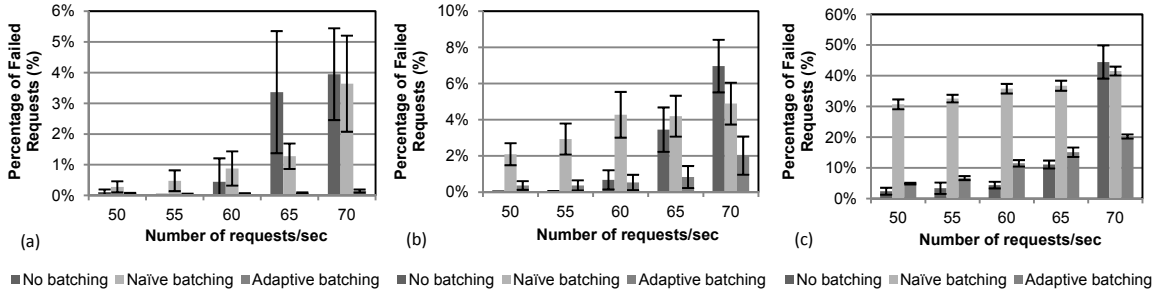

Figure 7.18. Percentage of Failed Requests with α in {0.7 (a), 0.8 (b), 0.9 (c)}.

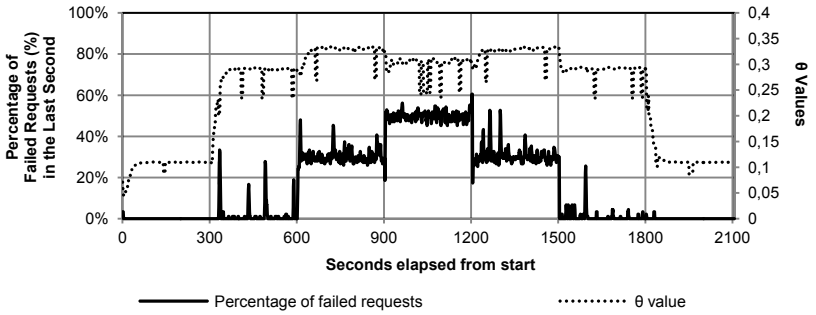

Figure 7.19. Percentage of Failed Requests and θ Values with Time-varying Workloads.

both the percentage of failed requests and the θ values for each second; for the sake of readability, we applied a smoothing filter in which each point is averaged with the previous four ones. Starting from the percentage of failed requests, SALES ensures very few failures when the request rate is lower than the processing one (first, second, sixth, and seventh timeslot). If the request rate is equal to the processing rate (third and fifth timeslot), we experience a percentage of failed requests around 30%. Finally, when the request rate is remarkably higher than the processing one (fourth timeslot), the percentage of failed requests becomes high and close to 50%. Considering the processing rate of 50 reqs/s, we would analytically expect an upper bound to the percentage of failed requests close to 30%: however, as also presented in the next section, the query drop policy discharges additional queries due to false positives, and the overall overload leads to increased late responses. In addition, let us note that the θ values carefully approximate and promptly follow real wireless network load. On the one hand, considering that each query and data is respectively around 1.5 and 4.5 KB long (the Java serialization introduces additional overhead to the real payload), and that ad-hoc links offer a 6 Mbps bandwidth, the estimated θ values are very close to the real ones: for instance, with 30 reqs/s, θ is close to 0.28, while the real value is around 0.24. On the other hand, θ promptly approximates the load every time the request rate changes. Some seconds are required to θ to adapt to the current load: this inertia is due to the distributed monitoring schema adopted for network load computation. In addition, we remark that, in the fourth timeslot and in contrast with the higher request rate, θ assumes lower values since many failed requests do not result in context data routing.

To conclude, by considering above results, we remark that the adaptive policy is able to effectively trade off average retrieval times with context distribution reliability. From the management viewpoint, the adaptive batching requires to distribute only lightweight wireless network load factors, periodically piggybacked on mobility beacons;

unfortunately, at the same time, it may lead to increased CPU overhead due to the higher number of scheduled task descriptors. By comparing the CPU load experienced when the adaptive batching policy is enabled with the one caused by no batching policy, we found that, in all the previous experiments, adaptive batching leads to an increased CPU overhead always smaller than 0.5%. Hence, considering the associated benefits, we think that the adaptive batching schema is always viable and feasible.

7.7.3. Query Dropping Evaluation

Query dropping is fundamental to control and limit the introduced CPU load, so as to better fit the current execution environment. Here, we present experimental results related with the different query dropping policies, namely naïve and adaptive query dropping, by also comparing them with the case when no query dropping is enabled.

In the following experiments, we always deployed 1000 context data sources on the CN, and we enabled context data repositories of 120 elements on intermediate nodes. Each node stores context data passing through it and, when the repository is full, a traditional LFU policy is applied to select the element to remove. We used LFU as replacement policy since, during preferential access patterns, it better fits the Gaussian distribution adopted by the traffic generator, thus reducing the requests relayed to upper levels and better highlighting the different CPU loads associated with different access patterns. Finally, considering that we use the unique SUN to simulate multiple clients, it has the local context data repository disabled: hence, all the queries are distributed, at least, up to the CUN.

All the experimental results presented in this section are obtained from a general test of 720 seconds, divided in 6 time slots of 120 seconds each, and with any slot with a different access pattern to simulate different scenarios. The SUN selects a target context data source, among the 1000 available on the CN, by using a uniform distribution in time slots {1, 3, 5} and a Gaussian distribution in time slots {2, 4, 6}. In addition, to simulate data access patterns with different degrees of preference, the Gaussian-based distributions of time slots {2, 4, 6} adopt respectively a standard deviation (S.D.) of {15, 30, 45}. We remark that Gaussian distributions simulate scenarios where mobile nodes in physical proximity require similar context data, such as localization-dependant access patterns; in this case, it is likely that context queries retrieve response from data repositories of close mobile nodes. Finally, all reported results are average values over 10 executions.

In the first set of experiments, we executed the above test with query drop disabled to

show that different access patterns lead to different CPU loads. In particular, we have used the SUN to emit {5, 10, 15} reqs/s, and we monitored the CPU load introduced on the CUN each second. For the sake of readability, we report average values of CPU load values over the last samples. Figure 7.20 (a) shows the CPU load during the test for different request rates. Of course, the higher the request rate, the higher the CPU load experienced by the CUN. In addition, it is worth remarking that CPU load depends on access patterns. When accesses are uniformly distributed, the CUN experiences higher CPU loads because it has to relay most of the queries to the BN. Instead, when accesses are distributed according to a Gaussian distribution, many queries find response from the CUN data repository, thus avoiding further query distributions. The lowest CPU load is associated with the second time slot where almost all the data are found on the CUN (Gaussian distribution with a S.D. of 15). In this test, no query drops occur because, as anticipated, both CUN and SUN drop policies have been disabled.

From Figure 7.20 (a), we remark that it is impossible to find a precise PQ_{MAX} value given a specific maximum CPU load. However, considering that the CDDI is a background service executed on mobile device, we can reasonably assume that it can introduce a maximum CPU load of 5%. Hence, from these initial tests, we conclude that a query processing rate of 10 reqs/s is appropriate to ensure a good number of satisfied queries, while keeping the final CPU load below 5%.

In the second set of experiments, we have enabled the naïve drop policy on the CUN with a static PQ_{MAX} equal to 10 to possibly keep the CPU load close to 5%. Figure 7.20 (b) and Figure 7.20 (c) represent respectively the average CPU load at the CUN and the percentage of satisfied queries for each second. Starting with Figure 7.20 (b), we note that all the main observations made for the experiments of Figure 7.20 (a) still apply. In addition, when query request rate is equal to 15 reqs/s, the CPU load does not remarkably increase since the CUN proactively drops queries in excess. Of course, it is possible to

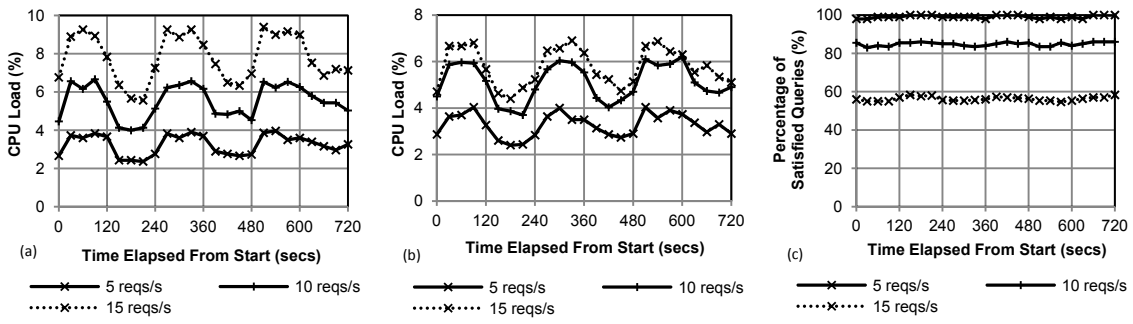

Figure 7.20. CPU Load with Naïve Query Drop Disabled (a) and Enabled (b), and Percentage of Satisfied Queries (c).

highlight a slightly higher CPU load due to the overhead introduced by query receive and dispatching. At the same time, in Figure 7.20 (c), when the request rate is higher than PQ_{MAX}, we obtain a percentage of satisfied queries that approximates the difference between the number of queries emitted by the SUN and the number of queries processed by the CUN. In fact, all the queries emitted by the SUN pass through the CUN that proactively discards them according to its own PQ_{MAX}. In addition, the naïve drop policy also leads to false positives when approaching PQ_{MAX}: when query request rate is equal to 10 reqs/s, the adopted drop condition leads to a percentage of satisfied requests close to 80%. Finally, from Figure 7.20 (b) and Figure 7.20 (c), we conclude that the naïve drop policy leads to unjustified query drops since unable to adapt to data access patterns. In fact, although the average CPU load is below 5% during Gaussian-based accesses (see Figure 7.20 (b)), the CUN keeps dropping a high number of queries (see Fig. Figure 7.20 (c)). Hence, the CDDI should automatically increment PQ_{MAX} during these periods to reduce dropped queries.

In the third set of experiments, we enabled our adaptive drop policy with a maximum CPU load of 5% and a monitoring period of 10 seconds. We recall that, due to the adopted monitoring period, $PQ_{MAX}(k)$ is adapted only every 10 seconds; this, of course, reduces the reactivity of the system in following fast changing CPU loads, but increases the stability of $PQ_{MAX}(k)$. By using the same traffic patterns introduced above and a request rates in {5, 10, 15} reqs/s, Figure 7.21 (a), Figure 7.21 (b), and Figure 7.21 (c) show respectively the CPU load, the percentage of satisfied queries for each second, and the values assumed by $PQ_{MAX}(k)$ during the whole test. Let us briefly note that, when query request rate is 5 reqs/s, the CPU load is always below 5% (see Figure 7.21 (a)). Hence, no queries are dropped (see Figure 7.21 (b)), while $PQ_{MAX}(k)$ is almost stable and close to 8 reqs/s (see Figure 7.21 (c)).

Instead, when query request rate is 10 reqs/s, our adaptive drop policy starts to adjust

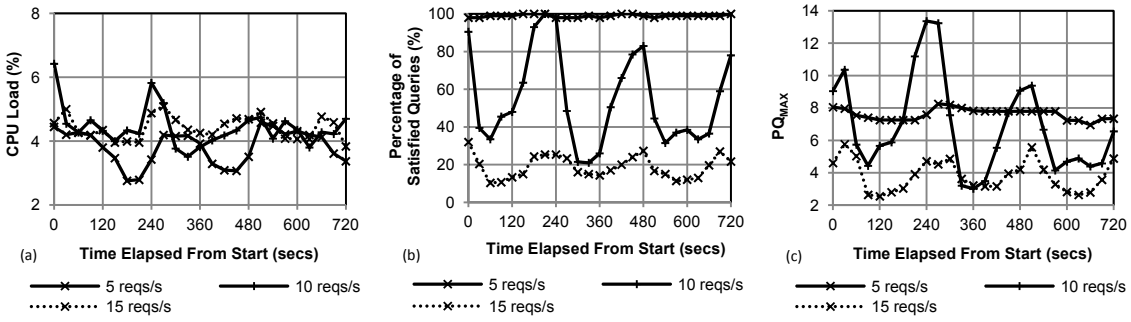

Figure 7.21. CPU Load (a), Percentage of Satisfied Queries (b), and PQ_{MAX} Values (c) with Adaptive Drop Policy Enabled.

$PQ_{MAX}(k)$. Starting from Figure 7.21 (a), we can note that, at the beginning of the test, the CPU load is higher than 5%. Hence, our adaptive drop policy reacts by reducing $PQ_{MAX}(k)$ (see Figure 7.21 (c)). This, in its turn, leads to a lower percentage of satisfied requests in Figure 7.21 (b). In the second time period, the data access pattern becomes Gaussian-based: hence, our adaptive drop policy increases $PQ_{MAX}(k)$ to reduce the failed requests. The same trend repeats every time the data access pattern changes. The main effect is that, differently from Figure 7.20 (c), the percentage of satisfied queries in Figure 7.21 (b) is time-dependent. It increases during Gaussian-based access patterns due to the reduced CPU load introduced by each query; of course, the highest value is reached at the end of the second time slot as it employs the lower standard deviation. In addition, in this test, during uniform access patterns, $PQ_{MAX}(k)$ is about 4 reqs/s. This could seem in contrast with the results presented in Figure 7.20 (a), where a CPU load of 5% is associated to a hypothetic request rate of 7.5 reqs/s. However, this is an unfair comparison since, in this case, the CUN exhibits a higher load due to additional query decoding and dispatching.

Finally, when the SUN emits queries with a request rate of 15 reqs/s, our adaptive policy reacts similarly. However, in Figure 7.21 (c), $PQ_{MAX}(k)$ tends to assume very low values for both uniform- and Gaussian-based access patterns. This is consequence of the fact that the CPU load introduced by the CDDI tends to be always higher than 5%. In other words, even if our adaptive drop policy reduces the processed queries, the CDDI keeps introducing a steady CPU load associated with message decoding and dispatching. However, as showed in Figure 7.21 (a), the adaptive drop policy enforces the final CPU load of 5%, hence, it achieves our main goal.

7.7.4. Evaluation of SALES on Android Devices

To better assess the feasibility of SALES CDDI on real-world resource-constrained mobile devices, as mentioned before, we ported our solution on the Android platform. In this section, we present experimental results from our real deployment, and we better highlight the management overhead introduced on mobile phones due to context data distribution.

In the following tests, we disabled intermediate repositories on all the mobile nodes, so to simulate the worst case scenario, namely all the queries have to be distributed up to the CN before retrieving context data. At the same time, as we have thoroughly evaluated SALES batching techniques in Section 7.7.2, here we consider data/queries distributed according to the no batching policy, in order to avoid additional failed requests due to

higher routing delays. Finally, all the experimental results report average values obtained over 10 executions.

We focused our following tests on three different distributed architectures, good representatives of the configurations that can be found in real deployments. The first one, "Laptop/WiFi", is a limited deployment CN-BN-CUN where the CUN is a laptop and connects to the BN through WiFi; it represents the best case scenario due to 1) shorter distance between the client and the CN; and 2) usage of a full-fledged laptop. The second one, "MobilePhone/WiFi", is similar to the previous one, but the CUN is an Android mobile phone: due to tighter resource constraints, we expect lower performance with respect to the first configuration. Finally, the third one, "MobilePhone/WiFi - MobilePhone/BT", extends the previous one to reach a full CN-BN-CUN-SUN configuration: both the CUN and the SUN are mobile phones that connect in ad-hoc through a BT link. We remark that we did not test WiFi ad-hoc links between mobile phones due to Android limitations.

The first set of experiments compares two key performance metrics: the average retrieval time and the percentage of failed requests (see Figure 7.22 (a) and Figure 7.22 (b)). We have considered request rates in {2, 4, 6, 8} reqs/s and, for each case, we show the average value and the standard deviation over a 5-minutes long test. The queries are always emitted by the CUN in the first two configurations, and by the SUN in the last one; in addition, the emitting node imposes a data retrieval time of 2 seconds. Since SALES estimates a worst-case distribution scenario made only by the nodes in the vertical path between the creator node and the CN, i.e., 2 hops when queries are emitted by the CUN and 3 hops when they are emitted by the SUN, it will impose $DRD_M = QRD_M = 500$ ms in the first case, and 333 ms in the second case. Then, since SALES applies an α factor equal to 0.7 to consider unforeseen delays introduced by local processing, the final DRD/QRD will be respectively equal to 350 and 233 ms: in conclusion, the routing process will try to

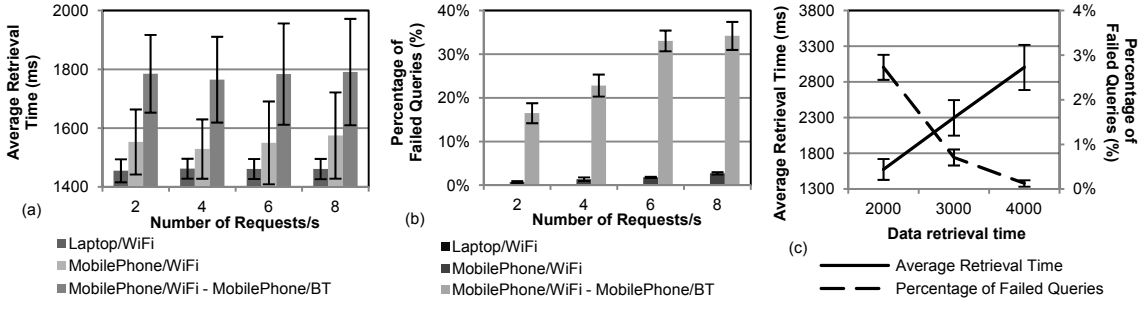

Figure 7.22. Average Retrieval Times (a) and Percentage of Failed Queries with Different Request Rates (b), and with Different Delivery Deadlines (c).

deliver data in 1400 ms, thus having 600 ms to cope with unexpected delays due to wireless channel congestion or local resources overload.

From obtained experimental results, the "Laptop/WiFi" case is the best one: it ensures the lowest average retrieval times and exhibits no failed requests; in fact, the usage of a full-fledged laptop with a standard JVM 1.6 makes the request rates of these tests negligible. Instead, the other two configurations experience increased average retrieval times and some failed responses since mobile phones introduce higher routing delays due to computational resource scarcity. In addition, the scarce memory reserved for an Android application, limited to a maximum of 24MB on the LG-P500, results in frequent GC collections, thus introducing additional delays (as also better detailed in our third experiment). Finally, the "MobilePhone/WiFi - MobilePhone/BT" scenario has the worst performance due to the higher number of transmissions and hops involved in the routing process; the surge of failed responses is also due to the fact that the CUN, by acting as router, suffers an increased memory pressure due to data/queries serialization.

In our second set of experiments, we aimed to clarify main causes of failed queries. By using the "MobilePhone/WiFi" test scenario and a fixed request rate of 8 reqs/s, Figure 7.22 (c) shows both the average retrieval times and the number of failed queries when the mobile phone requires data with a data retrieval time in {2000, 3000, 4000} ms. We remark that the percentage of failed queries decreases with higher data retrieval time values. Such behaviour is related with SALES routing delay mechanism: in fact, if data retrieval time is equal to 2000 ms, SALES has 600 ms to cope with unforeseen delays; instead, with a data retrieval time of 4000 ms, it has 1200 ms of remaining time. Hence, the CDDI can better recover unpredictable delays introduced by Dalvik GC and scarce computational resources, thus reducing the number of failed queries.

Finally, we have better observed the CDDI heap size via Android "*adb shell dumpsys meminfo*" command. We remark that the current implementation does not employ any particular optimization to address the heap fragmentation problem (see Section 7.6.4), so as to produce a worst case scenario where programmers do not consider Android peculiarities. In the "MobilePhone/WiFi" test scenario, we have monitored the Dalvik heap for a 20 minutes long test by using request rates in {2, 4, 6, 8} reqs/s. Figure 7.23 shows the heap size sampled once every second. Of course, the higher the request rate, the higher the heap size because of more frequent object allocations and increased heap fragmentation. We remark that the heap size of Figure 7.23 is the total heap size perceived by Android; this differs from the allocated heap (smaller and not shown in figure) that is

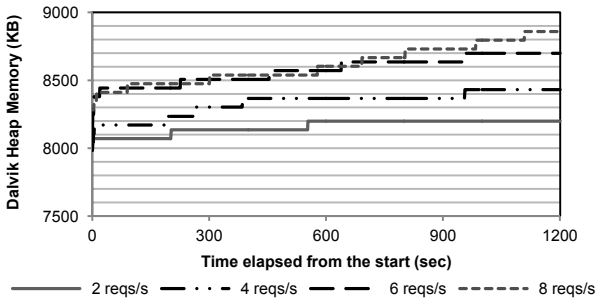

Figure 7.23. Dalvik Heap Memory during a 20 Minutes Long Test.

the fraction of the heap used to store referenced objects that cannot be released. For a particular request rate, the allocated heap size is almost constant during the whole test due to automatic GC mechanisms. Instead, although the request rate is constant, the memory dedicated to our CDDI client goes up over time. That represents a usual accumulation effect of many GC implementations: in our case, these effects are worsened by the heap fragmentation problem. Hence, that core issue must be taken into account before a production phase, also because, if the application reaches 24 MB, the Android runtime will automatically kill it to avoid slowing other external applications.

8. Context Data Distribution in Smart Cities Scenarios

Our SALES CDDI strives to effectively address the deployment of context-aware services in large-scale scenarios. It employs novel solutions, mainly based on the joint usage of heterogeneous wireless modes, cooperative context data repositories, and quality-based constraints, to improve system scalability and reliability. However, when we have to scale to very large deployments, such as the ones that we can find in smart cities scenarios, the design of suitable CDDIs introduces additional challenging issues that have to be properly addressed; amongst others, huge amounts of context data produced into the system have to be processed according to context-aware service needs.

This chapter focuses on the realization of large-scale context-aware systems, and introduces both fundamental issues and main directions in this research area. We extend our distributed architecture to include Cloud computing solutions, in charge of storing and processing the context data produced into the mobile system. We remark that this chapter does not share the common structure of the previous two ones; in particular, here we leave out the already discussed and well detailed issues related to context data distribution and management in mobile environments, in order to have more space for an in-depth discussion about Cloud computing and the advantages associated to their usage in smart cities scenarios.

The rest of this chapter is organized as follows. In Section 8.1, we detail the usage of Cloud computing solutions for CDDIs. In Section 8.2, we present the main challenges and management issues introduced by Cloud computing architectures. Then, since Cloud architectures need complex management infrastructures to efficiently deal with modern data center, in Section 8.3 we detail our Cloud management infrastructure, by presenting the core Virtual Machine (VM) placement problem. Finally, Section 8.4 presents an in-depth discussion about our original contributions, concerning network-aware VM placement, by introducing a new optimization problem, as well as heuristics to solve it.

8.1. Cloud Computing in CDDI

Large-scale city-wide scenarios feature thousands of sensors that continuously push new context data into the mobile system. State-of-the-art mobile devices are equipped with several onboard sensors, such as camera, GPS, and accelerometers, that continuously produce new data useful to characterize the current situation. In the smart city vision,

physical environments will be also equipped with sensors, e.g., temperature/humidity sensors and cameras, feeding new information directly into the fixed infrastructure. Consequently, to efficiently manage the storage and the processing of such large amounts of context data, we decided to adopt Cloud computing solutions [39, 47, 100].

Cloud solutions allow the rapid provisioning of scalable and reliable services, by means of distributed and virtualized hardware/software resources. The intrinsic scalability of such architectures, coupled with the possibility of provisioning computing resources only when required, makes them very suitable to store and process data coming from large-scale mobile systems. Modern Cloud solutions also exploit multiple data centers spread all over the world, thus enabling the dynamic provisioning of computing resources close to particular physical locations. Hence, they offer the computing power useful to realize new compelling context-aware scenarios in large-scale mobile settings, with the possibility of provisioning such resources closer to the point-of-attachments of the wireless infrastructures.

In our vision, the CDDI dynamically asks for computing resources to the Cloud, releasing them when no longer needed. The dynamic scaling of resources lets the CDDI require new computing resources when the context data to be processed increase; in fact, several conditions, such as time of the day and scheduled events (e.g., workshops and conferences), result in large fluctuations of the amount of context data pushed into the system. At the same time, the CDDI can dynamically reallocate computing resources between different services, for instance, to favour the processing of time critical data, and can automatically control the Cloud deployment to both release resources and possibly turn off not required physical servers, so as to eventually reduce the power consumption and the operational costs of the data center. All these interesting properties enable the rapid and efficient provisioning of context-aware services.

Although such vision is appealing as the CDDI can exploit the Cloud to effectively address context data storage and processing, Cloud solutions require complex management infrastructures to enable the dynamic and rapid provisioning of computing resources. The CDDI has to be aware of the increased complexities introduced by Cloud management, by carefully driving the reconfigurations associated with resource provisioning. Hence, in this chapter, we will focus on Cloud management aspects, with the main goal of highlighting how the CDDI should constraint and drive runtime Cloud reconfigurations.

8.2. Main Issues & Challenges

Cloud management is still a challenging task due to the novelty of models, technologies, and tools [122]. Currently, many industrial efforts have realized specific implementations capable of providing primitive Service Level Agreements (SLAs), such as number of CPUs, memory, and disk space allocated to each VM. Unfortunately, more advanced management operations, such as dynamic and automatic service scaling and reconfiguration facilities, useful to match the highly variable resource demands of large-scale context-aware services are still quite missing.

In general, Cloud solutions exploit virtualization techniques for the sake of VM consolidation, namely the provisioning of multiple VMs on the same physical host. A Cloud management infrastructure with the goal of VM consolidation must implement a proper placement function to detail final VM-to-host mappings. Since VMs will host real services used to process context data, VM placement is fundamental and can greatly affect the performance of executed services. Similarly, the dynamic scaling of such services is not straightforward, and can require tight interactions between the service and the Cloud management levels. Finally, it must be noted that, similarly to what happened in the SALES scenario, the transfer of important context data from the mobile to the fixed infrastructure, and vice versa, is an important issue due to tight bandwidth limitations of traditional fixed wireless infrastructures. Accordingly, in Section 8.2.1 and Section 8.2.2, we will detail the fundamental issues that have to be addressed in applying Cloud computing solutions to CDDIs.

8.2.1. Management Issues of the Cloud

A first and foremost Cloud management issue is to decide the placement of each VM in the data center, including decisions about co-locating more VMs on the same physical host. More formally, given a set of physical hosts equipped with finite resources and a set of VMs with resource requirements, the Cloud placement function has to find proper VM-to-host mappings that optimize a particular cost function. VM placement usually deals with conflicting goals, combined through different weighting factors according to how the Cloud provider ranks them; for instance, common cost functions minimize the number of turned on physical hosts, so as to reduce the operational costs of the data center, while keeping spare capacities to prevent frequent resource shortages. The Cloud placement function has to also consider multiple resource constraints, coming from limitations of physical hosts, to avoid placement solutions that would violate user SLAs. Hence, a

placement function must address two main directions: 1) an objective function, to rate how good a VM placement solution is; and 2) resource constraints, to avoid unfeasible VM consolidation solutions.

Starting with the objective function, Cloud providers usually tend to consolidate VMs on as few physical hosts as possible to reduce the power consumption of the data center, so to finally increase their economical revenue. Other different and heterogeneous goals can be defined. Load balancing between physical hosts is important to prevent that overload situations can easily lead to resource shortage; in addition, for the sake of reliability, a Cloud provider can introduce anti co-location goals, detailing sets of VMs that should not be placed on the same physical host, to prevent that a single host failure affects the availability of whole running services. With a slightly different perspective, when Cloud solutions are used to support CDDIs in mobile scenarios, we need VM placement solutions that allow the easy provisioning of additional computational resources, with no need of VM relocations, namely VM migrations useful to free resources in order to accommodate incoming requests.

Then, the Cloud placement function has to model and enforce resource constraints to reach meaningful solutions. Physical hosts have limited CPU, memory, and I/O capacities that have to be carefully considered to avoid low performance and resource saturation. In general, resource constraints are expressed with a host-level granularity; however, additional and more complex constraints can be associated with the whole data center. For instance, from a networking viewpoint, both the network topology of the data center and the adopted routing schema greatly affect the real bandwidth available between physical hosts [123]; that complicates the placement problem since a particular solution, even if feasible from a host-level perspective, can result in link saturations. In addition, network traffic between co-located VMs is carried out locally, by means of in-memory message passing mechanisms; on the one side, this saves precious network resources but, on the other one, it increases CPU and memory overhead, with final runtime effects difficult to estimate and dependent on both hypervisors and device drivers. Since context data processing services usually introduce high network traffic, the Cloud placement function has to carefully consider such increased CPU/memory overhead to prevent physical host saturation.

Finally, placement optimization is based on the assumption that VMs present repeatable patterns on particular time scales, e.g., time of the day, period of the year, and so on. Hence, a pre-filtering phase is usually introduced to select an important, though

reduced, subset of VMs that present repeatable patterns along resource requirements. By exploiting such property and historical data, the Cloud placement function can better decide the VMs to consolidate, thus reducing the possibility of reaching unfeasible placements due to time-varying resource requirements. We remark that, if we consider the usage of Cloud architectures for context processing, such characteristic of predictability is often valid: produced context data usually depend on the people gathered in a particular physical place, and this is intrinsically associated with both the specific hour and the day of the week.

Hence, although the adoption of Cloud computing architectures for CDDIs complicates both the design and the deployment of such systems, we claim the feasibility of such solutions to increase system scalability, while ensuring an effective and efficient usage of the Cloud physical resources.

8.2.2. Bridging together the Mobile and the Fixed Infrastructure

Although the adoption of Cloud computing solutions presents clear advantages, the real-world realization of such solutions has a fundamental issue, mainly related to the transfer of the context data between the mobile and the fixed infrastructure. Context-aware services need to exchange huge amounts of data, coming from sensors either deployed on mobile nodes or on fixed infrastructures. All these data have to be transferred through bandwidth-constrained wireless infrastructures, thus introducing a high traffic due to the system scale.

Apart from our SALES CDDI [124], several academic works considered the opportunistic usage of mobile devices as data carriers, so to distribute data to close devices through ad-hoc links. Few works investigated cooperative content sharing services based on the joint usage of infrastructure-based and ad-hoc communications [46, 125]; similarly, hybrid architectures, such as the one adopted by HiCon [30], have been recently proposed for cooperative context data distribution in large-scale mobile systems. Even if these relevant efforts are now producing interesting results, we claim the need of more sophisticated techniques capable of orchestrating the context data distribution into the whole system.

Above all, it is worth remarking that, in city-wide scenarios, the usage of cellular infrastructures for the continuous upload and download of context data would probably lead to prohibitive network traffic and economical costs [41]. The context data distribution has to be carried out principally by ad-hoc links between mobile devices. In addition,

differently from SALES scenario, here we need routing protocols that, in a delay tolerant fashion [38, 126, 127], progressively transfer the context data close to specific collection points, such as WiFi hotspots that allow not expensive data offloading. In other words, while SALES focuses more on synchronous data retrieval operations, where context queries are distributed to collect interesting context data as soon as possible, in this case context distribution operations can present less tight deadlines, and we can exploit these relaxed time constraints to better coordinate mobile nodes in the whole system.

Hence, an interesting research direction is the usage of Cloud architectures to process important data associated with mobile nodes, e.g., mobility traces, context data associated with device owner, and so forth, to effectively reconfigure the context distribution process at runtime. Collected data about mobile devices can be processed into the Cloud, by exploiting similar mechanisms to the ones adopted for context data processing. Then, suitable reconfiguration commands can be sent to mobile devices to properly reconfigure the context data distribution mechanisms.

8.3. Cloud Management Infrastructures

A Cloud management infrastructure is a complex software stack solution that requires a deep understanding of several and heterogeneous aspects, spanning from monitoring schema to virtualization technologies and networking architectures. Currently, several commercial products already offer comprehensive management infrastructures, and some of them, such as Amazon EC2 [128], offer primitive mechanisms to perform dynamic resource provisioning and automatic service scaling into the Cloud data center. For the sake of readability, we leave out detailed descriptions about general management infrastructure for Cloud computing, and we focus only on the main phases useful to understand the remainder of this chapter (see Figure 8.1).

First, the Cloud management infrastructure has to gather a wide set of indicators, including VM load metrics, power consumption of physical hosts and network elements, and so forth. Since data centers usually include physical elements belonging to different vendors, it is important to integrate with and use different monitoring protocols. Second, the Cloud management infrastructure has to decide the optimal VM placement solution. This decision is very complex from a computational viewpoint, since it usually includes the resolution of multi-criteria optimization problems with a large solution space, that can be limited by user SLAs, host computational resources, power consumption, etc. Finally, if a better VM placement is found, the Cloud management infrastructure has to dynamically

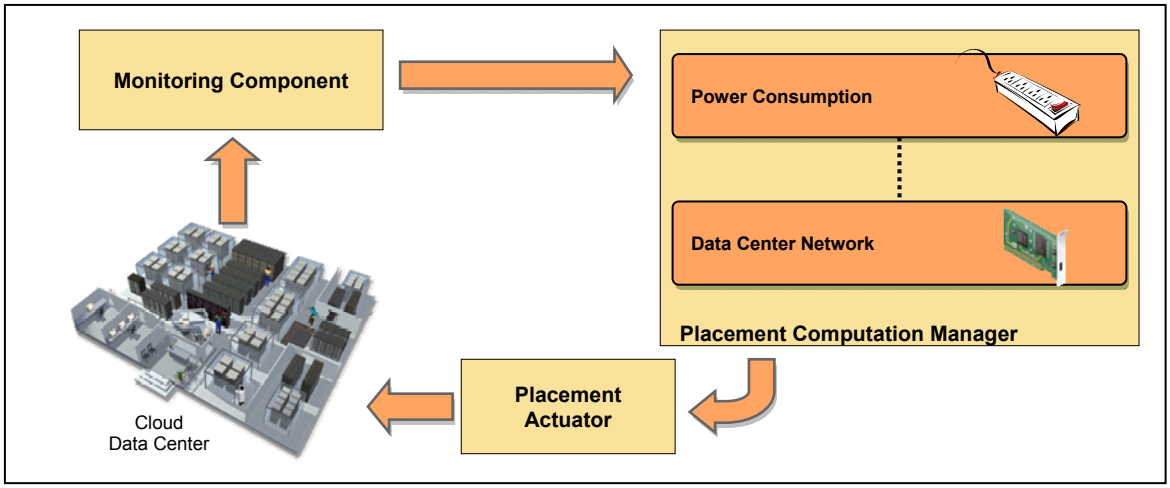

Figure 8.1. Logical Architecture of a Cloud Management Infrastructure.

reconfigure the data center, by also detailing a suitable plan of VM relocation operations, and by taking over hypervisor and network path reconfigurations.

To deal with such complexity, a Cloud management infrastructure usually adopts a three-stage architecture: the first stage collects monitoring data from the Cloud; the second one exploits collected data to calculate a new and more suitable VM placement, if available; finally, if this is the case, the third stage calculates the VM relocation plan and applies suggested changes to the Cloud data center. Hence, a Cloud management infrastructure (see Figure 8.1) consists of a *Monitoring Component*, a *Placement Computation Manager*, and a *Placement Actuator*, arranged in a pipeline where each stage uses, as input, the output of the previous one. The Monitoring Component gathers both system and service level information about used resources, and makes them available to the next stage. The Placement Computation Manager exploits both monitoring information and user SLAs to check whether a better VM placement exists; as showed in Figure 8.1, it contains additional sub-modules to consider specific resource dimensions, such as networking and power. Finally, if a better placement exists, the Placement Actuator takes care of executing the VM migration plan and reconfiguration operations.

Between aforementioned components, our work primarily focused on the Placement Computation Manager, namely the component in charge of detailing the single VM-to-host associations. Between different aspects, we considered the introduction of network-aware constraints and optimization goals into the VM placement problem; Section 8.4 presents our work in this research area.

8.4. Network-aware Placement

All the VMs hosted in the same Cloud data center intrinsically share network

resources. Even if Virtual LANs (VLANs) are usually adopted to create and separate logical networks, mainly for security reasons, both network links and switching elements are shared between all VMs. Hence, traffic demands of different Cloud customers interfere among themselves, and this can lead to reduced service performance. In addition, both the modeling and the introduction of network constraints are complex tasks to be addressed.

In the past years, different works considered network-aware VM placement for Cloud data center. Network resource constraints are usually considered with a host-level granularity, by enforcing that the aggregate traffic from/to a physical host is lower than the maximum capacity of the network interface. However, this is a very simplistic assumption since Cloud data center usually adopts complex hierarchical topologies where pairs of hosts experience different time-varying bandwidth, depending on routing schema and conflicting traffic. In [123], authors considered the problem of VM placement with the goal of reducing the aggregate traffic into the data center. Since the proposed placement problem is NP-hard, authors propose a new heuristic, based on clustering algorithms, to solve real-world instances in reasonable time. While this work assumes that inter-VMs traffic demands are static and well-defined, another important work consolidates together VMs with uncorrelated traffic demands [129]: it introduces network constraints with a host-level granularity, and places together VMs by ensuring that local network capacity is violated with a probability lower than a specific threshold. Since traffic demands are expressed through stochastic variables, the proposed VM placement problem is a stochastic bin packing problem. Then, in [130], authors consider the placement of applications made by a computation and a storage part, with the main goal of reducing aggregated network traffic. However, similarly to [123], authors do not consider constraints on single links, thus potentially leading to unfeasible placement solutions; in addition, since traffic demands are only between the computation and the storage part of each application, the resulting problem is not as hard as the one in [123]. Finally, in [131], authors consider Cloud data centers with server and storage virtualization facilities, and strive to increase load balancing at multiple layers, including servers, switches, and storage. They propose a new placement algorithm that considers multiple resource dimensions from these layers; since the proposed algorithm is a multi-dimensional knapsack problem, it is largely inspired by the famous Toyoda method [132].

Although these are extremely valid works, we think that additional research has to be done in this area. First, Cloud data centers feature non-trivial network topologies that connect physical hosts through multiple paths for the sake of scalability and reliability

[133-135]. Second, since dynamic multi-path routing protocols are usually introduced to exploit the full available bandwidth between hosts, traffic demands routed along particular network paths change at runtime. Finally, Cloud data centers host heterogeneous services that can lead to extremely different runtime traffic patterns, with a high variability due to either unpredicted request spikes or service-dependant operations, e.g., database replication.

Consequently, to address these issues, we propose a new network-aware VM placement problem, namely Min Cut Ratio-aware VM Placement (MCRVMP). MCRVMP targets virtualized data centers and takes into account constraints on both local physical resources (CPU and memory) and network related ones. As regards the latter, it considers both complex network topologies and dynamic routing protocols, and exploits the notion of network graph cuts to express associated constraints. Most important, starting from the realistic assumption that inter-VMs traffic demands are time-varying, MCRVMP strives to minimize the maximum load ratio over all the network cuts, so as to find VM placement solutions that, by having spare capacity on each network cut, have higher probability of absorbing unpredicted traffic variations.

In the following subsections, we present additional details about our network-aware placement component. In Section 8.4.1, for the sake of clarity, we introduce common data center network topologies. Then, in Section 8.4.2, we present our new network-aware VM placement problem while, in Section 8.4.3, we detail our heuristics to solve it. Finally, in Section 8.4.4, we introduce experimental results to support the technical soundness of our MCRVMP proposal.

8.4.1. Data Center Network Topologies

Modern Cloud data centers feature heterogeneous services that can lead to very different communication patters, from one-to-one to all-to-all traffic matrices. In addition, network topology greatly affects the maximum achievable bandwidth with specific traffic patterns. To accommodate this wide range of service requirements, a plethora of solutions has been presented in the research literature for specific classes of services [133-136].

Broadly speaking, most of the network topologies for data centers share a three-tier architecture [136]. The lowest *access tier* contains the real physical hosts that directly connect to access switches. The intermediate *aggregation tier* contains aggregation switches that connect together access switches, so to allow more localized network communications among hosts. Finally, the highest *core tier* contains core switches that

connect aggregation switches; this tier also includes the gateways for the traffic with the outside of the Cloud data center. Despite the multitude of proposals, we remark that three main topologies are emerging as standard-de-facto solutions: Tree, Fat-tree, and VL2 (see Figure 8.2).

Tree-based networks are appealing due to their simplicity of wiring and reduced costs [136]. Unfortunately, such topologies suffer of very low reliability and scalability bottlenecks. In fact, a single link failure completely disconnects the network in two sub-trees. In addition, moving toward the tree root, such topologies are usually oversubscribed, meaning that, in the worst case scenario, the aggregate traffic coming from one side of the tree cannot be transferred to the other one due to limited link bandwidth. We also recall that tree-based topologies are usually the result of adopted routing protocols; in fact, even if the physical network topology is a graph, the usage of both VLANs and spanning tree routing algorithms leads to tree-like logical networks.

Fat-tree is a three-tier topology extensively based on bipartite graphs. Basic building block of this topology is the so-called pod (see dotted areas in Figure 8.2), namely a collection of access and aggregation switches connected in a complete bipartite graph. Each pod is connected to all the core switches, but links are evenly spread between the aggregation switches contained into the same pod; hence, this leads to a new (not complete) bipartite graph between aggregation and core switches. Fat-tree topology assumes that all the switches have the same number of ports, and this greatly limits topology scaling. If N is the number of port per switch (Figure 8.2 shows an example topology for N equal to 4), the resulting Fat-tree has N pods, each one containing $\frac{N}{2}$ aggregation switches and $\frac{N}{2}$ access switches. Each pod connects to $\frac{N^2}{4}$ servers and to $\frac{N^2}{4}$ core switches. The main advantage of this topology is the availability of multiple paths between each pair of hosts: in fact, $\frac{N^2}{4}$ disjoint paths can be used to route the traffic between two physical hosts; unfortunately, Fat-trees are extremely expensive due to the very high

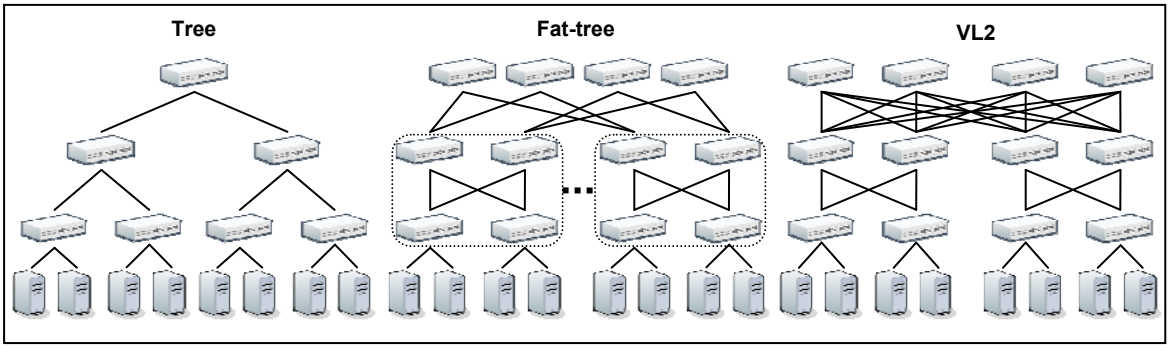

Figure 8.2. Common Data Center Network Topologies.

number of switching elements and links.

Finally, VL2 is also a three-tier architecture sharing important similarities with Fat-tree [133]. Main difference is that VL2 adopts a complete bipartite graph between core and aggregation switches, and not between aggregation and access switches. VL2 exploits a new routing schema, called Valiant Load Balancing, that forces packets received at the access switches to be forwarded up to the core layer. Hence, even if two hosts are connected to different access switches that, in their turn, connect to the same aggregation switch, packets between them are always forwarded first to a randomly selected core switch, and then back to the real destination. This routing schema is based on the fact that, if traffic patterns are unpredictable, the best load balancing between available links is obtained by randomly selecting a core switch as intermediate destination [133].

Our MCRVMP strives to minimize the maximum cut load ratio, in order to potentially support time-varying traffic demands with reduced packet droppings and no additional VM relocations. General network graphs can present an exponential number of cuts, but several of them are not useful in MCRVMP problem formulation. Above all, we are interested in network cuts that partition the set of hosts in two non-empty and connected subsets, as they are bottlenecks for the traffic demands between VMs placed on different sides of the cut. In the remainder, we call them *critical cuts*; a critical cut with a load ratio close to 1 implies that all the traffic demands carried through it have a little degree of variability before leading to dropped packets.

Starting from simple tree-based networks, we have a critical cut for each network link. The removal of one link partitions the network and leads to two different and connected subsets of hosts. Hence, associated MCRVMP constraints can be easily expressed. Instead, both Fat-tree and VL2 topologies present several bipartite graphs that lead to a higher number of network cuts, thus making the cuts analysis more complex. However, as better explained in the following, in the case of MCRVMP, we can reduce Fat-tree and VL2 topologies into *equivalent tree networks*. Now, we give the main guidelines behind our topology transformations; we decided to omit additional algorithm details as they can be easily derived.

Let us focus on the Fat-tree topology in Figure 8.3 (a). The Fat-tree contains homogeneous links with equal capacity C. For each pod, under the assumption of dynamic routing, we can study only a limited number of cuts: in particular, for each access switch, we can define a network cut (see NC_1 and NC_2) containing all the uplinks toward the aggregation tier; then, we can define an additional cut (see NC_3) by removing all the

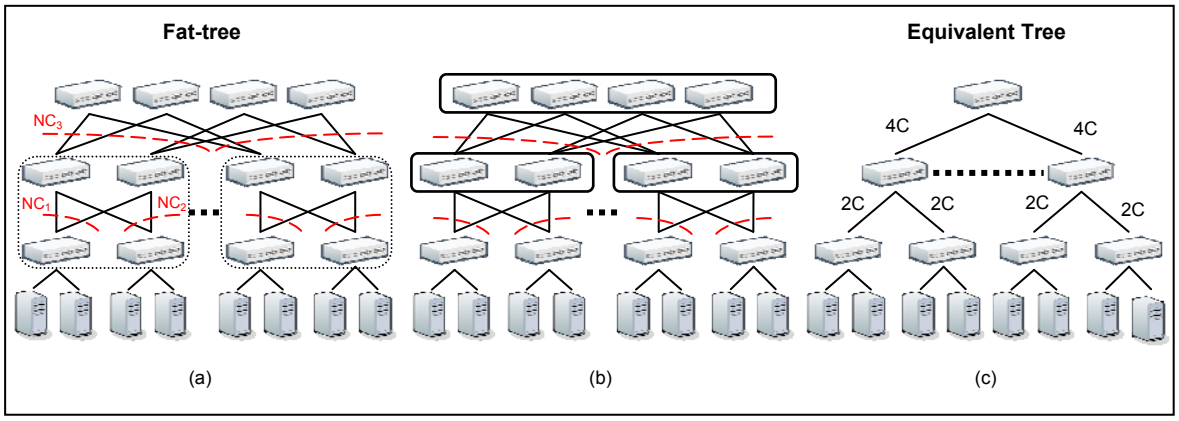

Figure 8.3. Fat-tree and VL2 Transformation in Equivalent Tree.

uplinks going out of the pod. By iterating these rules on all the pods, we can find the important network cuts for a Fat-tree topology (see Figure 8.3 (a)). In other words, we can transform all the switches in the boxes highlighted in Figure 8.3 (b) with virtual switches, one for each box; then, all the links crossed by a network cut can be represented by a virtual link with equivalent capacity. Hence, the Fat-tree topology of Figure 8.3 (a) can be transformed into the equivalent tree of Figure 8.3 (c). Similarly, for a VL2 topology, we introduce a cut for each access switch by removing all the uplinks. Then, depending on the actual wiring, we identify bipartite graphs of access and aggregation switches: for each one of them, we introduce an additional cut containing all the links between it and the core tier. Of course, for both Fat-tree and VL2 topologies, we consider the network cuts associated with the links between physical hosts and access switches (not showed in Figure 8.3 for the sake of readability).

Once the critical network cuts have been identified, each one of them can be replaced, for the sake of MCRVMP, by a single link with equivalent capacity. Applying this transformation to the highlighted network cuts in Figure 8.3 exemplifies how Fat-tree and VL2 topologies can be reduced to an equivalent tree. After, MCRVMP is solved on the equivalent tree: focusing on the critical cuts, both the obtained placement solution and the network cut values apply for the initial network topology. Thanks to this property, in the remainder we consider only tree-based networks, and all the presented heuristics apply also to Fat-tree and VL2 topologies.

8.4.2. MCRVMP Problem Formulation

MCRVMP considers that traffic demands are usually time-varying since several factors, i.e., specific time-of-the-day, periodic tasks, etc., can deeply affect real network traffic demands. Hence, it tries to find placement solutions resilient to traffic variations in

deployed services, by minimizing the maximum load ratio over all the network cuts. In a more formal way, *MCRVMP finds a VM-to-host placement that, while respecting resource constraints on host CPU, host memory, and network cuts, minimizes the maximum network cut load*.

In finer details, MCRVMP regards the placement of VMs on physical hosts belonging to a virtualized data center. We consider a set of *n* host $H=\{h_i\}_{i=1,\dots,n}$ and a set of *m* VMs $=\{vm_j\}_{j=1,\dots,m}$. Each host h_i is described by resource capacities $\langle CPU_{CAP}, MEM_{CAP}\rangle$, while each VM vm_j has resource requirements $\langle CPU_{REQ}, MEM_{REQ}\rangle$. We use a traffic matrix T to represent the average traffic rate between each pair of VMs; the element t_{ij} is the traffic rate from vm_i to vm_j. Finally, X is an $m \times n$ matrix of binary variables used to represent placement information; hence, the element x_{ij} is 1 if vm_i is placed on host h_j, 0 otherwise.

The data center network topology is a tree (in case of Fat-tree or VL2, graph transformations presented in Section 8.4.1 are applied first). Hence, any critical network cut contains only one link, and has a total capacity equal to the capacity of the single contained link. Moreover, each network cut partitions the set of hosts H in two disjoint subsets, H_1 and H_2; for the sake of clarity, H_1 always contains the hosts of the sub-tree originating from the link endpoint farther from the original tree root. In a valid placement solution, the aggregate traffic flowing from VMs placed in H_1 to VMs placed in H_2 must be lower than the cut capacity (the same applies to the traffic from H_2 to H_1). Finally, a Cut Load Ratio (CLR) vector contains all the cut load ratio values. For each cut, CLR has two elements: the first one is the ratio between the traffic flowing from H_1 to H_2 divided by the cut capacity; the second one is similar, but considers the traffic flowing from H_2 to H_1. Hence, CLR size is two times the total number of cuts: the entries for H_1 to H_2-traffic are stored in the first half of the vector and all the others in the second part.

To express critical network cuts, we introduce a cut matrix C where each row represents a network cut and each column a host; the element c_{di} is 1 if h_i belongs to H_1 when the d^{th} cut is considered. Hence, C matrix represents topology information, and has to be generated off-line by computing all the critical network cuts. This step is simple for a tree topology: it only requires to remove one link each time and to express the C row associated with the two host partitions H_1 and H_2. For instance, Figure 8.4 shows the cut matrix C associated with a simple binary tree (the cuts associated with host-to-access switch links are not shown for the sake of readability); the label of each row represents the link contained into the network cut, namely the link removed to generate the associated

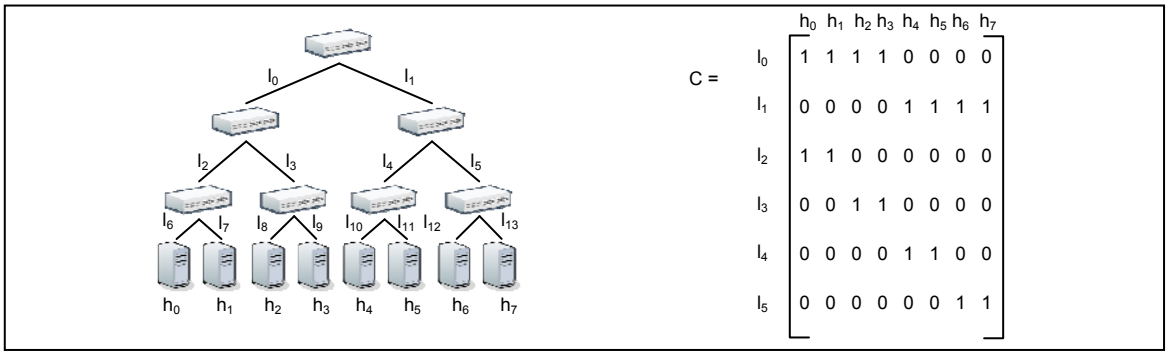

Figure 8.4. Cut Matrix C for a Simple Binary Tree.

cut. By using the cut matrix C, the total traffic rate from H_1 to H_2 (respectively, from H_2 to H_1) over the d^{th} cut is the d^{th}-element of the diagonal of the matrix $[C \times X^T \times T \times X \times (1\text{-}C)^T]$ (respectively, $[(1\text{-}C) \times X^T \times T \times X \times C^T]$). This accounts only for the traffic between VMs; the traffic from/to the gateway, situated at the tree root, is also added according to the VMs placed in H_1 (not shown in the following model for the sake of readability).

Hence, MCRVMP is formally expressed through the following integer quadratic programming model:

$$\min\left(\max_{c \in 1\ldots 2 \times N_{cut}} CLR_c\right) \tag{8.1}$$

$$\sum_i vm_i.CPU_{REQ} \times x_{ij} \leq h_j.CPU_{CAP} \quad \forall j \tag{8.2}$$

$$\sum_i vm_i.MEM_{REQ} \times x_{ij} \leq h_j.MEM_{CAP} \quad \forall j \tag{8.3}$$

$$CLR_c = \begin{cases} \dfrac{[C \times X^T \times T \times X \times (1-C)^T]_{cc}}{CAP_c}, & \forall c \in \{1, \ldots, N_{CUT}\} \\ \dfrac{[(1-C) \times X^T \times T \times X \times C^T]_{cc}}{CAP_c}, & \forall c \in \{N_{CUT}+1, \ldots, 2 \times N_{CUT}\} \end{cases} \tag{8.4}$$

$$CLR_c \leq 1 \; \forall c \in \{1, \ldots, 2 \times N_{CUT}\} \tag{8.5}$$

$$\sum_j x_{ij} = 1 \quad \forall i \tag{8.6}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, \forall j \tag{8.7}$$

Formulas (8.2) and (8.3) enforce CPU and memory capacities at each single host. Formulas (8.4) express the aggregate traffic flowing on each network cut. Formulas (8.5) enforce feasible solutions from the network point-of-view. Formulas (8.6) avoid the same VM to be placed on different hosts, while formulas (8.7) define x_{ij} as binary variables. Formula (8.1) expresses MCRVMP goal, namely to minimize the maximum cut load ratio.

Finally, we remark that the MCRVMP problem formulation presented here is not meant to be applied on-line, namely to serve new VM requests in a data center that already

contains placed VMs. In fact, here we focus on finding the optimal MCRVMP solution, while leaving out VM relocation costs; those have to be considered in on-line formulations of our optimization problem, in order to prevent frequent and expensive VM migration operations.

8.4.3. Solving MCRVMP

MCRVMP is an NP-hard problem that can be optimally solved only for very small and unrealistic problem instances. Hence, to make it suitable for real-world Cloud scenarios, we designed two placement algorithms, called *2-Phase Connected Component-based Recursive Split (2PCCRS)* and *Greedy Heuristic (GH)*, with different tradeoffs between solution quality and execution time.

Both our placement algorithms take advantage of the fact that Cloud scenarios feature Connected Components (CCs) of VMs that exchange data only between themselves or with the external gateway, such as the case where multiple VMs of the same customer run a three-tier web application. Hence, it is reasonable to cluster VMs in CCs so that, by considering a CC as a single VM to be placed, we can reduce problem complexity. In addition, as tree networks suggest the use of recursive algorithms where intermediate placement sub-problems partially fix VM-to-host assignments, 2PCCRS uses this property to further reduce the solution space; this leads to a reduced applicability since 2PCCRS can be applied only if the network topology is (or can be transformed into) a tree. In the following subsections, we clearly detail our two placement algorithms.

8.4.3.1. 2PCCRS Placement Algorithm

Our first placement algorithm, 2PCCRS, uses mathematical programming techniques to solve the MCRVMP problem, and has two main properties. First, it adopts a two-phase approach: the first phase places CCs to sub-trees, while the second phase expands them to place actual VMs on physical hosts. Second, it is recursive: in both phases, it exploits the tree network structure to define and solve smaller problem instances on one-level trees. That allows 2PCCRS to deal with MCRVMP complexity, by reducing the number of VMs, hosts, and network cuts at each placement step.

With a closer view to algorithm details, the first phase places CCs to force associated VMs to be mapped in specific sub-trees. 2PCCRS processes the matrix T to cluster VMs in CCs, stored into the set $CC=\{cc_d\}_{d=1,\dots,t}$. Each cc_d is associated with a resource requirement vector $\langle CPU_{TOT}, MEM_{TOT}, IN_{TOT}, OUT_{TOT}\rangle$ that expresses aggregated CPU and memory requirements, and total download/upload traffic from/to the gateway. Since

CCs do not exchange traffic among them, during this phase we have to only model traffic demands between each CC and the gateway. According to the recursive approach, 2PCCRS starts from the tree root, and solves an initial placement problem on a one-level tree by considering a set of virtual hosts $VH=\{vh_z\}_{z=1,\ldots,q}$, one for each child node the real tree root has. Each virtual host represents the total capacity available in the sub-tree rooted at the real node, hence, associated resource capacities are equal to the sum of all child nodes capacities. Every time a one-level placement problem is solved, the set of CCs is partitioned among available virtual hosts. Then, 2PCCRS recursively solves sub-problems associated with the one-level trees where $\{vh_z\}_{z=1,\ldots,q}$ are roots, and this process repeats until we solve all the placement sub-problems associated with real access switches. At the end of this phase, we have CCs placed on particular node of the initial topology; in the second phase, VMs of such CCs have to be placed in the sub-tree rooted at the specific intermediate node.

For the sake of clarity, let us consider the simple example of Figure 8.5. We have a full binary tree made by 8 hosts and a total of 16 VMs to place. From T processing, 2PCCRS identifies $CC=\{cc_0, cc_1, cc_2\}$, respectively containing 9, 5, and 2 VMs. 2PCCRS starts by considering the sub-problem P1 (step a), and tries to place the CCs on the VHs. Due to resource constraints, only cc_1 and cc_2 are placed; in addition, they are placed on different virtual hosts to reduce the cut load ratio. Then, 2PCCRS solves both sub-problem P2 and sub-problem P3, associated with the aggregation switches of the real topology. However, cc_1 is bigger than the capacity of considered VHs, hence, it is not moved from the tree root of P2 (step b). Instead, cc_2 can be placed in one VH, hence, it is pushed toward one of the real access switches (step c). At this point, the first phase terminates by supplying CC-to-network switch relationships; for instance, due to obtained results, all the

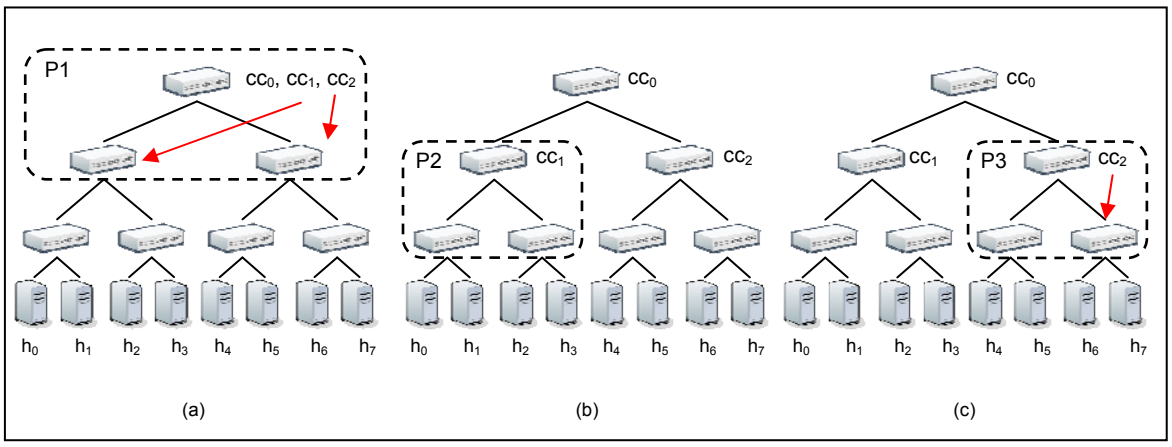

Figure 8.5. 2PCCRS Placement Computation Example – First Phase.

VMs belonging to cc_2 will be placed between h_6 and h_7.

Once clarified this general process, we note that the placement problems solved by 2PCCRS during the first phase differ from MCRVMP as detailed in Section 8.4.2. In fact, due to resource constraints, it could be impossible to place all the involved CCs at each step: hence, $\sum_z x_{dz} = 1 \quad \forall d$ has to be relaxed in $\sum_z x_{dz} \leq 1 \quad \forall d$ to have feasible results. However, such relaxation is not useful since the solution with all the CCs not placed is feasible and ensures the minimum worst case cut load ratio, namely 0. Hence, we associate each cc_d with a penalty traffic equal to the average inter-VM traffic demand between contained ones. If a cc_d is not placed, we add its penalty traffic, as well as its traffic to/from the gateway, to all network cuts; in this way, if possible, a cc_d will be always placed to reduce the maximum cut load ratio. More formally, the placement sub-problem solved at each step is represented by the following integer linear mathematical model (formula (8.8)-(8.17)):

$$\min\left(\max_{c \in 1 \ldots 2 \times N_{cut}} CLR_c\right) \tag{8.8}$$

$$\sum_d cc_d.CPU_{TOT} \times x_{dz} \leq vh_z.CPU_{CAP} \; \forall z \tag{8.9}$$

$$\sum_d cc_d.MEM_{TOT} \times x_{dz} \leq vh_z.MEM_{CAP} \; \forall z \tag{8.10}$$

$$isCCPlaced_d = \sum_{\forall z} x_{dz} \quad \forall d \tag{8.11}$$

$$penaltyT = \sum_{\forall d} (1 - isCCPlaced_d) \times cc_d.penaltyTraffic \tag{8.12}$$

$$isBelowCut_{dc} = \sum_{\forall z \text{ in } H_1(c)} x_{dz} \; \forall d, \forall c \tag{8.13}$$

$$CLR_c = \begin{cases} \frac{\sum_{\forall d}[cc_d.IN_{TOT} \times (1 - isCCPlaced_d + isBelowCut_{dc})] + penaltyT}{CAP_c}, & \forall c \in \{1, \ldots, N_{CUT}\} \\ \frac{\sum_{\forall d}[cc_d.OUT_{TOT} \times (1 - isCCPlaced_d + isBelowCut_{dc})] + penaltyT}{CAP_c}, & \forall c \in \{N_{CUT} + 1, \ldots, 2 \times N_{CUT}\} \end{cases} \tag{8.14}$$

$$CLR_c \leq 1 \; \forall c \in \{1, \ldots, 2 \times N_{CUT}\} \tag{8.15}$$

$$\sum_z x_{dz} \leq 1 \quad \forall d \tag{8.16}$$

$$x_{dz} \in \{0, 1\} \quad \forall d, \forall z \tag{8.17}$$

In the second phase, 2PCCRS splits CCs to place real VMs. This phase adopts a recursive approach similar to the one of the previous phase, but it solves real MCRVMP

problem instances. At each step, it considers all the VMs associated with CCs that have been placed during the first phase in the considered tree root. VH capacities are adjusted according to the CCs placed in the sub-tree rooted at the current vh_z; this means 1) subtracting the aggregate CPU and memory requirements associated with placed CCs; and 2) adding traffic demands to/from the gateway to consider the aggregate traffic demands coming from placed CCs. In all the subsequent steps, each sub-problem considers a set of VMs made by both VMs inherited from the father node and VMs belonging to CCs placed at the current vh_z during the first 2PCCRS phase. By following a recursive approach, 2PCCRS keeps solving intermediate sub-problems up to the leaves, where we finally have VM-to-host associations.

For the sake of clarity, Figure 8.6 presents an example of the second 2PCCRS phase, consequence of the initial placement performed in Figure 8.5. At the first placement sub-problem P1 (step a), 2PCCRS has to place all the VMs associated with cc_0, previously associated with the tree root. It considers that cc_1 and cc_2 are placed in sub-trees, respectively rooted at the first and at the second aggregation switch, by 1) subtracting their aggregate resource consumptions from VH capacities; and 2) by adding traffic demands to/from the gateway, so to mimic the real traffic introduced by cc_1 and cc_2. Due to resource constraints, 3 VMs, namely vm_0, vm_1, and vm_2, are placed on vh_1, while the remaining ones on vh_2. At the second sub-problem P2 (step (b)), 2PCCRS has to place a set of VMs equal to the union of VMs coming from P1, namely {vm_0, vm_1, vm_2}, and associated with cc_1, placed during the first step; hence, P2 will place {vm_0, vm_1, vm_2, vm_9, vm_{10}, vm_{11}, vm_{12}, vm_{13}}. A similar reasoning is applied to solve all the sub-problems rooted at the other network switches; at the end, the placement sub-problems associated with the access switches will give the VM-to-host associations.

Focusing on 2PCCRS complexity, there are few important things to highlight. First,

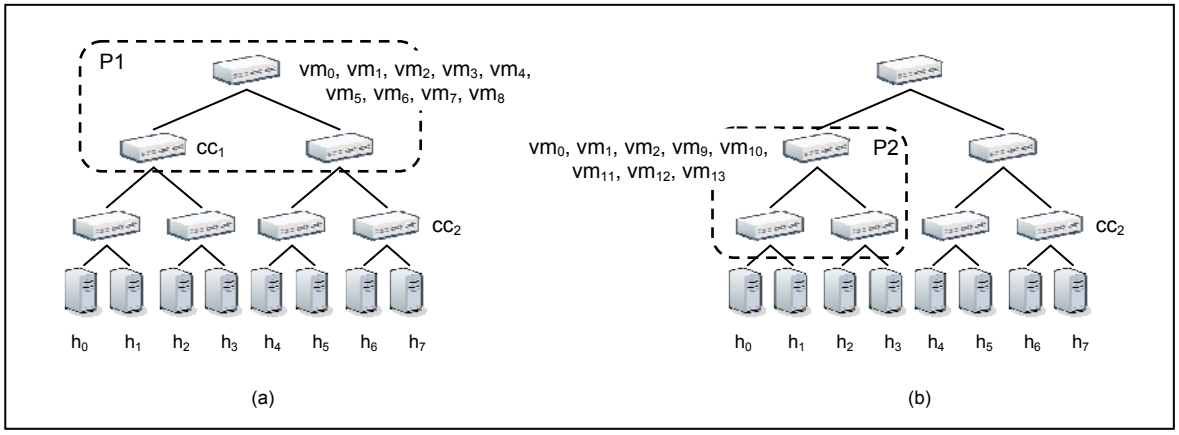

Figure 8.6. 2PCCRS Placement Computation Example – Second Phase.

since CCs do not have traffic demands between themselves, the placement sub-problems of the first phase are integer linear (not quadratic) programming problems; also, the number of CCs is usually much smaller than the one of VMs. Second, thanks to the first phase, the second phase of 2PCCRS usually has to solve small problem instances. For instance, if we consider the placement sub-problem associated with the tree root, 2PCCRS does not initially consider all the VMs associated with CCs that, during the first phase, have been placed in VHs rooted at aggregation and access switches.

8.4.3.2. GH Placement Algorithm

Our second placement algorithm, GH, completely leaves out mathematical programming techniques and greedily places VMs on available hosts. Differently from 2PCCRS, where intermediate sub-problems fix VMs to be placed in sub-trees, GH places each VM individually, thus having more freedom during placement computation. In brief, GH consists of two main phases: the first one ranks all the traffic demands, while the second one exploits them to place VMs on available hosts.

Let us anticipate some notations that we use in the remainder. Since VMs are iteratively placed, it is possible that a placed VM has traffic demands from/to VMs not placed yet. A traffic demand of such kind is called *floating* in respect of all network cuts, since we cannot establish a-priori which cuts it will influence. If a traffic demand has both end-points placed, we define it *committed* because it is possible to clearly understand which cuts it affects. Finally, during placement computation, a traffic demand is committed by a VM-to-host placement if its status changes from *floating* to *committed* due to current placement operation.

In the first phase, GH extracts the CCs out of the traffic matrix T. After, it ranks them to find the ones more difficult to split from the point of view of MCRVMP objective function. Toward this goal, it orders all the traffic demands by decreasing values, and associates each cc_d with an accumulator whose value is the sum of the relative positions occupied by the traffic demands belonging to cc_d in the ranked list. Intuitively, the higher the accumulator value, the higher the number of big flows contained into cc_d, hence, the bigger the variations of the cut load values during cc_d splitting will be. Finally, GH orders CCs by decreasing accumulator values, and then, following this order, extracts the traffic demands; for each CC, demands are considered in decreasing order.

In the second phase, GH iteratively processes the ranked traffic demands. For each traffic demand, it initially selects the VM to place. If both the VMs involved in the traffic demand have been already placed, it skips to the next demand; if only one of them has

been placed, it considers the remaining one; finally, if both VMs are not placed yet, it considers the one that, after the current placement operation, would commit the higher number of demands. Then, GH filters all the hosts to consider only the ones having enough resource capacities to accommodate the current VM. It iteratively tries to place the VM on each feasible host, while evaluating all the network cut values. At the end, the VM is placed on the host that will lead to the minimum value of the maximum cut load value. GH iterates above steps until all the VMs are placed.

However, the evaluation of network cut load values is possible only after a full VM placement has been determined, and not while the placement is ongoing; in fact, once committed, floating demands can greatly affect network cut load values. To approximate final cut load values in an ongoing manner, we merge floating and committed traffic demands. Let us focus on a particular network cut within a partial placement: in the best case, all the floating traffic demands will be routed to hosts belonging to the same partition, thus leading to a final total traffic over the cut equal to the already committed demands; instead, in the worst case, all the floating traffic demands will be routed to hosts belonging to the other partition, thus leading to a final traffic equal to the sum of committed and floating demands. The latter situation is likely to happen when the floating traffic demands originate from a partition with residual capacities close to zero; in fact, in that case, subsequent VMs would be likely placed on the opposite partition, thus routing floating traffic demands over the cut.

Hence, during ongoing VM placement, we estimate the final traffic over the network cut as a weighted sum of committed and floating demands. We differentiate traffic demands flowing from one partition to the other, and vice versa. For each direction, the aggregate traffic routed in the partial placement contains committed flows, with weighting factor 1 since they will surely appear in the final solution, and floating ones, with a weighting factor proportional to the worst case ratio of residual capacities. Finally, the obtained value, divided by the cut capacity, is the final cut load value considered by GH.

8.4.4. MCRVMP Experimental Results

We evaluated our placement algorithms along two main directions. First, in Section 8.4.4.1, we focus on MCRVMP-based placement computation by comparing random, optimal, 2PCCRS, and GH solutions. Then, in Section 8.4.4.2, we validate the technical soundness of proposed placement algorithms by NS2-based simulations: we generate synthetic traffic demands and we show that obtained placement solutions are indeed able

to tolerate time-varying traffic demands.

8.4.4.1. Comparisons between Placement Algorithms

Here, we compare our two heuristics, 2PCCRS and GH, by focusing on placement quality and solving time. To better assess our proposals, we consider two additional algorithms. The first one, called Random (RND), randomly generates VM-to-host assignments; it is useful to compare MCRVMP-based placements with a network-oblivious one. The second one, called Optimal (OPT), uses a mixed integer programming solver to solve the entire MCRVMP problem; hence, it finds the optimal solution, i.e., the VM placement that minimizes the maximum cut load ratio. Due to the associated complexity, experimental results for the OPT algorithm are available only for extremely small problem instances.

All the following experimental results are associated with a data center made by a pool of homogeneous hosts having the same capacity for CPU and memory resources. In addition, all VMs have equal CPU and memory requirements; hence, due to capacity constraints, each host in the pool can accommodate the same number of VMs. The data center network is always a fully balanced tree with link capacity of 1 Gbps. We execute our heuristics on a physical server with CPU Intel Core 2 Duo E7600 @ 3.06GHz and 4 GB RAM, and we exploit IBM ILOG CPLEX as mixed integer mathematical solver to compute OPT solutions and solve the intermediate steps of 2PCCRS. ILOG is always configured with pre-solve and parallel mode enabled; due to hardware limitations, it exploits a maximum of 2 threads during solving. Finally, all the reported experimental results are average values of 10 different executions; in addition, we report standard deviation values to better assess the confidence of our results.

One crucial aspect is the modeling of the traffic matrix T. We have to produce CCs but, at the same time, we need to test our heuristics with different T as the total number of considered traffic demands greatly affects problem complexity. Hence, we generate T taking into account three main parameters: 1) CCs size; 2) traffic patterns between VMs of the same CC; and 3) rate of the traffic demand, in terms of Mbps. For the sake of readability, we focused our evaluation on one challenging and realistic case study. For CCs size, we consider them distributed according to a uniform distribution. Then, traffic demands between VMs in the same CC are randomly generated with a probability lower than 1, and with rate following a Gaussian distribution (mean = 5 Mbps, standard deviation = 0.5 Mbps). Also, each CC has a VM with both upload and download traffic demands to the gateway, with rates generated according to another Gaussian distribution

(mean = 2 Mbps, standard deviation = 0.2 Mbps).

In the first set of experiments, we focused on small problem instances in order to be able to compare our heuristics with the OPT algorithm. The adopted network topology is a three-level binary tree; 24 VMs have to be placed on 8 physical hosts, under different traffic matrices. Here, we consider CC sizes according to a uniform distribution in [1; 8]; inter-VMs traffic demands are randomly generated with a probability in {0.5, 0.75}. Figure 8.7 (a), Figure 8.7 (b), and Figure 8.7 (c) respectively show the maximum cut load value, the average cut load, and the placement computation time. Focusing on the first graph (see Figure 8.7 (a)), both 2PCRRS and GH reach maximum cut load values very close to the OPT algorithm, while RND is the worst one as it does not consider traffic demands. In Figure 8.7 (b), we note that, to minimize the maximum cut load ratio, OPT produces an average link load higher than the ones produced by 2PCCRS or GH. Hence, at the end, our heuristics usually carry less traffic into the data center than OPT, but they lead to higher maximum cut load ratios. Finally, we evaluated placement computation time: RND, 2PCCRS and GH have execution times close to zero for these little problem instances; OPT, instead, as it can be seen in Figure 8.7 (c), presents extremely high computation times. We also note that computation time increases as the number of communicating pairs increases. Those times confirm that OPT is not feasible for real-world Cloud scenarios; in addition, OPT exhibits placement computation times with very high standard deviation values. Hence, for specific problem instances, namely the ones with several small CCs, the solver is able to reach the optimal solution quickly, while instances with dense traffic matrix T are extremely complex to solve. In brief, OPT computation time is not only very long, but also difficult to predict.

In the second set of experiments, we focused on a wider network deployment by using a fully balanced quaternary tree with 64 hosts. In this case, we increment the number of VMs (from 2x to 20x the number of hosts), to compare heuristics scalability; as regards traffic matrix, CC sizes follow a uniform distribution in [1; 16], while associated traffic

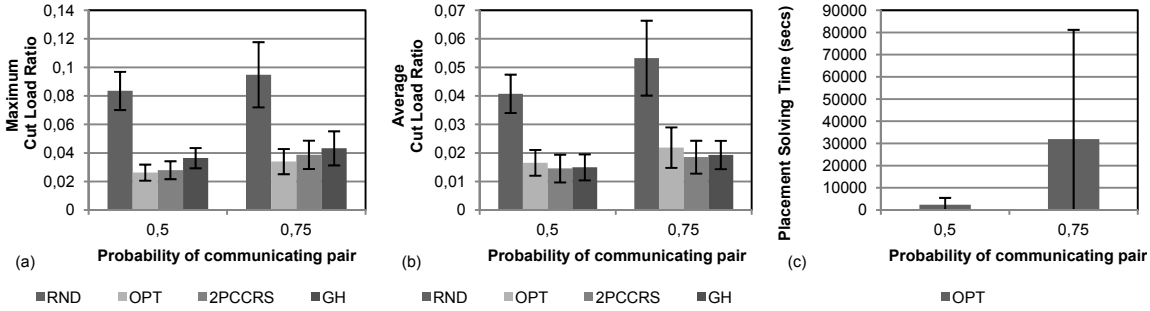

Figure 8.7. Placement Algorithms Results for a Small Data Center of 8 Hosts.

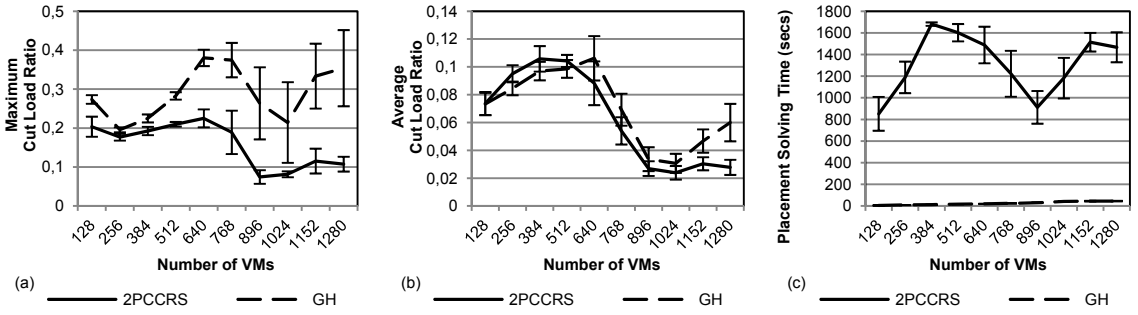

Figure 8.8. Placement Algorithms Results for a Data Center of 64 Hosts.

pairs are generated with a probability of 0.75. Figure 8.8 (a), Figure 8.8 (b), and Figure 8.8 (c) respectively show the same set of results used in the previous case for this new scenario. RND algorithm is not showed since it was able to reach feasible placements (with maximum cut load ratios higher than 0.9) only for the simpler case of 128 VMs; apart from that, it always reached unfeasible placements due to cut values higher than 1, hence, we decided neither to consider nor to show these results. Focusing on Figure 8.8 (a) and Figure 8.8 (b), we remark that 2PCCRS and GH reach similar results for smaller number of VMs; then, starting from 640 VMs, 2PCCRS always performs significantly better than GH. From Figure 8.8 (b), we note that 2PCCRS also favours lower average link loads. Although 2PCCRS leads to better VM placement solutions, it has high computation times. In Figure 8.8 (c), GH presents placement computation times that increase almost linearly with the number of VMs (in the worst case, it computes the placement in about 50 seconds). Instead, 2PCCRS computation time is higher due to the usage of mathematical programming techniques With aforementioned numbers of VMs, solving time increases remarkably, as each 2PCCRS placement step actually tries to find the optimal solution; at the same time, the solver typically finds very good results in the very first optimization steps, and then it only obtains limited improvements when run for longer time spans. In our experiments, we limit maximum placement computation time to 1800 seconds because we found that this total solving time ensures a good tradeoff between solution quality and placement computation time. Similarly to the previous scenario, the execution times of the solver are not predictable and depend on the specific problem instance; the case with 384 VMs was actually the one with longest solving time.

In the last set of experiments, we tried to evaluate the scalability of our heuristics as the data center grows. We fixed a number of VMs per host equal to 10, and we scaled the data center topology from 64 to 343 hosts by considering fully balanced trees; hence, we considered from 640 to 3430 VMs. As regards traffic matrices, we used the same

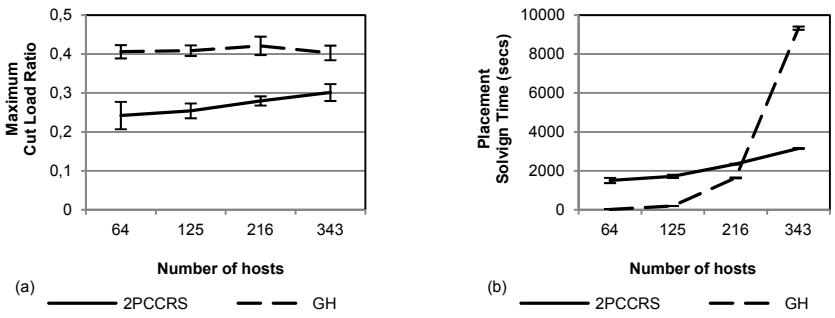

Figure 8.9. Placement Algorithms Results for Different Data Center Sizes.

parameters of the previous experiments. Figure 8.9 (a) shows the maximum cut load values achieved by our heuristics: we can see that 2PCCRS performs better than GH. In addition, Figure 8.9 (b) shows the total placement computation times of the two heuristics. While GH is faster than 2PCCRS for small data center sizes, it is much more sensible to topology scaling; this is mainly due to the fact that, at each placement step, GH considers all the hosts and all the network cuts. For instance, in the worst case, GH considers 343 hosts and 56 cuts at each VM to place; instead, at each one-level tree to solve, 2PCCRS considers only 7 network cuts and 7 virtual hosts. Even if we limit the solving time to 1800 seconds, 2PCCRS can reach very good solutions.

We conclude that both 2PCCRS and GH can reasonably solve MCRVMP with different tradeoffs between solution quality and placement computation time. 2PCCRS always reaches lower maximum cut load ratios, and scales better with topology size, while GH is significantly faster for small data center topologies.

8.4.4.2. Placement Validation with NS2 Simulations

We used NS2 to better assess the resilience of MCRVMP-based placement solutions under time-varying traffic demands. Due to space constraints, we focus on the case of 64 hosts and 128 VMs (see Figure 8.8). We selected that specific case since 2PCCRS and GH have different maximum cut load ratios (see Figure 8.8 (a)), but similar average cut load ratios (see Figure 8.8 (b)); in this way, we aim to find performance indicators that mainly depend on the maximum cut load ratios. Then, for each placement solution, we remove traffic demands between VMs co-located on the same host; each remaining demand is mapped in NS2 through an UDP source/sink pair. For each placement solution, we run 10 simulations with different seeds, thus having a total of 100 runs for each case study; in the remainder, we show average values and standard deviations of all the considered simulations. Finally, each NS2 simulation lasts 3600 seconds.

Each source produces a constant traffic rate according to the demand contained in the

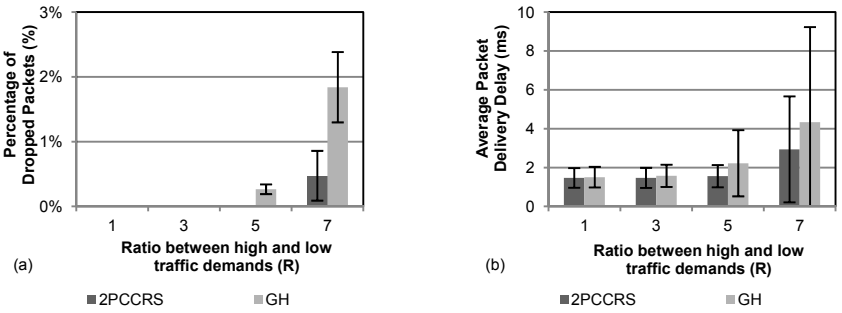

Figure 8.10. Percentage of Dropped Packets (a), and Average Packet Delivery Delay (b) in NS2 Simulations.

traffic matrix, by emitting UDP packets of 60KB each. Then, after D_{LOW} seconds, the source increases the traffic rate to R times the nominal value, and this increased demand lasts for D_{HIGH} seconds. This process repeats for the whole simulation, thus having normal and high traffic rates interleaved by D_{LOW} and D_{HIGH} times. D_{LOW} (respectively, D_{HIGH}) values are produced by a Gaussian distribution with mean of 200 seconds and standard deviation of 20 seconds (respectively, 100 seconds and 10 seconds for D_{HIGH}).

Figure 8.10 (a) and Figure 8.10 (b) respectively show the percentage of dropped packets and the average packet delivery delay for R in {1, 3, 5, 7}. Both 2PCCRS- and GH-based placements can absorb traffic demands up to three times the nominal values with no dropped packets. When R is 5, GH-based placements start experiencing dropped packets. In fact, from Figure 8.8 (a), we note that such solutions have a maximum cut load ratio close to 0.3; hence, when R is 5, the worst case cut (and the ones with similar load values) will be likely to be congested. Similarly, 2PCCRS-based placements experience dropped packets when R is 7. Finally, Figure 8.10 (b) shows that 2PCCRS-based placements have average packet delivery delays lower than GH-based ones, due to the less loaded network cuts.

To conclude, statistically speaking and considering that average cut load ratios are similar between 2PCCRS- and GH-based VM placements, performance improvements of 2PCCRS over GH are mainly consequence of the reduced maximum cut load ratio; hence, MCRVMP-based placements increase the capability of absorbing time-varying traffic demands.

9. Essential Contributions

In the previous chapters, we presented our work on CDDIs for large-scale mobile systems. Our case studies showed the applicability of our logical model in three different and significant deployment scenarios; also, we thoroughly evaluated our proposed solutions by means of both real deployments and network simulations. Let us anticipate, as a first general conclusion, that CDDIs for mobile systems present a great deal of complexity when both scalability and quality-based constraints need to be achieved, but quality-based constraints can enable runtime system management to dynamically adapt involved data distribution functions.

In this chapter, we remark and detail all main technical achievements and the future research directions highlighted by this thesis. In Section 9.1, by exploiting the experimental results showed in the previous chapters, we present a short summary of our main findings. Then, in Section 9.2, we draw our current research work and we present future research directions to the work presented in this dissertation.

9.1. Main Thesis Findings

CDDIs for mobile systems have to seamlessly integrate and interoperate with heterogeneous networks and mobile devices, toward the correct delivery of the context data into the mobile system. CDDIs complexity depends on both adopted network deployment and quality levels to guarantee. Although context-aware services are interesting from the industrial viewpoint, since they can attract more mobile users through extended service offerings, at the current stage we can find only a rather limited diffusion and we think this lack stems from the fact that clear models and definitions of CDDIs for large-scale mobile systems are still missing. Hence, our main contributions can be of use toward a better understanding of the area along the following directions.

Above all, we have analyzed the main mechanisms involved in CDDIs for mobile systems, by detailing and presenting a comprehensive logical model with associated design guidelines and choices. To better assess the technical soundness of our CDDI logical model, we have considered a large set of pre-existing context provisioning infrastructures in mobile systems; our survey work, to be published in the ACM Computing Surveys journal [5], supports the validity of our logical model and draws important tradeoffs between network deployments, context data distribution functions, and

quality constraints. We remark that, for the sake of readability, in this dissertation we have omitted our in-depth categorization of pre-existing infrastructures for context distribution; interested readers can refer to our survey work [5].

Then, we have focused on the real-world usage of our design guidelines by means of three significant case studies (presented in Chapter 6, Chapter 7, and Chapter 8). We have shown that the adaptation of the context data distribution function, properly guided and constrained by quality contracts, is fundamental to foster system scalability. The first RECOWER project focuses on important quality-based constraints and how to exploit them toward the main goal of increasing the number of successfully routed data. Our work has followed two principal research directions. In the first one (see Section 6.5.1), we have investigated the usage of quality constraints to dynamically reconfigure context data caching on mobile devices. Our approach, based on the introduction of differentiated quality classes, can increase context data availability and average data up-to-dateness; at the same time, it introduces an extremely contained management overhead, required to exchange quality classes between mobile nodes in physical proximity. In the second one (see Section 6.5.2), by using query/data routing delays, we have proposed an adaptive query flooding protocol with the main goal of reducing context query replication into the MANET. Our protocol, based on the exchange of lightweight management data, can effectively reduce the number of distributed queries and message collisions, thus increasing final context distribution reliability.

Instead, SALES considers the enforcement of our quality constraints in hybrid network deployments, where a fixed infrastructure can be used to store and supply access to context data. This second project exemplifies how the physical locality principle is useful to partition the context data into the distributed architecture, toward the main goal of keeping context data as close as possible to potentially interested consumers. In this case, our work followed three main directions. In the first one (see Section 7.5.1), we have considered the caching of relevant context data, in order to reduce the number of requests relayed to the fixed infrastructure. We have proposed an adaptive caching approach that, by considering access patterns and context data cached in physical surroundings, can effectively reduce the total number of requests sent to the fixed infrastructure. In the second one (see Section 7.5.2), we have extended the use of the routing delays to introduce batching techniques, so as to reduce the total number of wireless channel accesses. Our adaptive batching approach effectively reduces wireless contention, by only requiring the exchange of small load indicators of wireless network interfaces,

piggybacked in node beacons. Finally, in our third direction (see Section 7.5.3), we have considered that mobile devices present tight CPU limitations, and we have proposed an adaptive query drop policy that dynamically enforces maximum CPU usage limitations. The proposed adaptive query drop approach can quickly adapt to time-varying access patterns, thus increasing final context data availability. We recall that SALES evaluations have been also conducted through a real wireless testbed; at the same time, we have also realized an Android-based implementation of our solutions, to account for routing delays and management overhead introduced by real-world mobile devices.

Finally, we moved to large-scale settings where we adopted Cloud-based solutions to handle the huge amounts of context data produced by mobile infrastructures. As the CDDI can dynamically ask for additional computational resources, while releasing them when no longer needed, we focused mainly on the management aspects of the Cloud infrastructure. We have introduced a new network-aware VM placement problem (see Section 8.4.2), as well as heuristics to solve real-world problem instances in reasonable times. Our simulation results show that our placement solutions can effectively absorb time-varying traffic demands, thus increasing the stability of the VM placement solution. Finally, we remark that, although we focused more on Cloud management, as highlighted also in the next section, we are pursuing new research directions that will include Cloud-driven runtime adaptations of the distribution function.

With our real case studies, we have also tested the validity of our CDDI logical model and design choices. Obtained experimental results have confirmed that our solutions and design guidelines, such as joint exploitation of heterogeneous wireless standards and modes at the network deployment, distributed data caching, and so forth, can effectively increase system scalability under quality-based constraints. From the context data management viewpoint, both data caching and replication mechanisms are useful to exploit and enforce locality principles, with the main goal of avoiding heavy context data exchange from/to the fixed infrastructure. All these mechanisms should use local (e.g., access frequencies) and distributed (e.g., number of copies in the physical area) attributes to trade off context data availability with introduced overhead. Moreover, as showed through our Android-based implementation, all such mechanisms have to be resource-aware to prevent excessive overhead on resource-constrained mobile devices.

To conclude, in this thesis work we strived to reach a balance between CDDI models/architectures/design choices and their own applicability in real-world settings. By pursuing these directions together, we aimed to better support the validity of our

theoretical work, and to foster the widespread adoption of such data distribution mechanisms in the research community. Let us remark that our CDDIs have been downloaded by several research groups around the world; we hope that the availability of such prototypes, coupled with the possibility of easily modifying our data distribution protocols, can push toward more complete and systemic research works in this research area. We feel that this dissertation can become a seed to nourish a fruitful development in quality of context-aware system diffusion.

9.2. Future Research Directions

Although this work has focused on a selection of few and important research directions, several other directions still deserve further investigation. Focusing on the specific context distribution function, we think that several mechanisms needed in distributed, scalable, and QoC-based CDDIs are still widely unexplored. Here, some current principal research directions we intend to pursue are:

QoC Frameworks Definition - Although several research works already considered QoC [3, 4, 7, 23-25, 124], the intrinsic ambiguity of this concept has not promoted a general and widely accepted definition. To the best of our knowledge, general QoC frameworks, capable of helping service designers to understand QoC representation, sensing, and runtime usage, are still missing. Although some QoC parameters, e.g., data up-to-dateness, can be easily applied to all context data, different context aspects may require more complex efforts. Data-specific parameters are difficult to standardize, since strictly related with represented context aspects; on the bright side, they can enable finer and more useful adaptations. For instance, considering localization as part of physical context, many solutions in literature, such as MiddleWhere [69], use a quality attribute called resolution. Such attribute captures the expected maximum difference between real and sensed localization data; as localization errors strictly depend on the adopted localization technique, many solutions agree upon the usage of the maximum possible error, ensured from the localization technology, to quantify resolution. However, for other context aspects (computing, physical, time, and user), such a general agreement on data-specific parameters is difficult to achieve. For instance, if we consider co-located users as part of the user context, there is no widely accepted quality attribute useful to characterize possible differences between real and sensed values. In addition, since different systems can adopt different sensing strategies (e.g., based on APs associations, on received beacons between devices, …) and different aggregation techniques (e.g., history-based,

probabilistic, ...) to estimate co-localized people, it is almost impossible to agree on a single quality attribute, similarly to what happened for localization. Hence, while general QoC parameters are available in literature, additional research is required to define data-specific QoC parameters.

Context Data Aggregation and Filtering Operators - At the context data management layer, two functions, namely aggregation and filtering, deserve also additional research work. In our opinion, aggregation techniques currently lack of efficient methods to handle QoC data attributes. Such attributes are fundamental to prevent the injection of erroneous aggregated context data; at the same time, the design of aggregation algorithms, useful to quantify QoC parameters of derived context data, is also challenging and, to the best of our knowledge, not well investigated into the research literature. Hence, further studies should aim at defining proper aggregation algorithms able to combine context data and QoC parameters. Moving to filtering techniques, they are used to foster system scalability by suppressing not important data transmissions. Of course, they affect perceived QoC since, by limiting exchanged data, context-aware services have more chances to use stale and invalid context information. Change-based techniques, namely those ones that suppress data transmissions until the latest transmitted value bears some similarity constraints with the current data value, are appealing as they ensure an upper bound to the maximum error between current and received context data values. Also, when context data assume predictable values, we can use filtering operators and history-based integration techniques to let mobile devices locally estimate current context data values, thus avoiding expensive context data transmissions. Although few research works have already tried to address the problem of context data forecasting with the main goal of reducing network data traffic, for instance, by exploiting Kalman filters forecasting [111], we think that additional research is required to make such approaches able to scale to thousands of sensors and mobile nodes. In fact, forecasting techniques usually introduce increased CPU and memory overhead on resource-constrained mobile devices; hence, although valid works already exist in literature, additional research should study the relationships between QoC degradation and the cost of filtering techniques.

Adaptive Context Data Dissemination - As presented in both Section 4.4.2 and our survey work [5], at the current stage several CDDI solutions exploit a context data distribution schema that only relies on one specific approach, i.e., flooding-/selection-/gossip-based. At the expense of more complex implementations, hybrid solutions, based on the joint usage of different dissemination algorithms, can lead to increased runtime

performance. For instance, if we consider a network deployment that can rely on a fixed wireless infrastructure, the CDDI can exploit 1) a selection-based approach to ensure context access; and 2) a flooding-/gossip-based approach to replicate data, so as to reduce context access time and distribution reliability. Instead, if the network deployment is a MANET, the CDDI can use 1) a selection-based approach with tight physical constraints (for instance, in the two-hops neighborhood) to disseminate only required data; and 2) a gossip-based approach to enable context data visibility in far away areas. Above all, flooding- and gossip-based dissemination algorithms are very promising. Even if flooding-based schemas present scalability issues, they are suitable if flooding is constrained by locality principles; in small-scale distribution, data flooding algorithms can address distribution with high availability, null state on mobile nodes, and reduced response times. Gossip-based approaches trade off scalability with delivery guarantees; the control of the probabilistic nature of gossip-based protocols is an interesting research direction. As regards this specific point, we remark that valid results have been obtained in the close DTN research area. For instance, both HiBOp and Habit show that user social state and relationships are good hints to drive gossip decisions [127, 137]; similarly, CAR demonstrates that low-level time context information, namely inter-contact times and frequencies of contacts, leads to good solutions as well [126]. Although these protocols are extremely valid when applied to DTNs, we think that additional research is required to apply them at the context data distribution function, where 1) communications are usually from one producer to multiple consumers; and 2) the interests of the context data consumers can present a high degree of variability due to mobility. Finally, toward the main goal of adopting and adapting different dissemination algorithms at runtime, additional research works should be directed toward the definition of meaningful attributes useful to 1) drive the selection of the proper dissemination algorithms; and 2) adapt their runtime behaviour to maximize system scalability.

Since above adaptive solutions can introduce heavy management overhead, to elaborate mobility traces and context requests gathered from thousands of mobile nodes, here we remark the significance of Cloud architectures as real enablers of such scenarios. In fact, as detailed in Chapter 8, the CDDI can temporarily offload monitoring data from mobile devices to a Cloud, while paying such computational resources on a pay-per-use basis. The high computational power ensured by a Cloud will enable the processing of such data in a reasonable time, thus allowing subsequent adaptations of context data distribution protocols in order to improve systems scalability under quality constraints.

10. Conclusions

The widespread adoption of mobile devices and wireless communications is pushing toward the realization of novel context-aware services characterized by the capability of adapting at runtime according to current conditions. Several services require context-aware capabilities to ensure correct service provisioning; such context information can also span multiple aspects, ranging from local computational capabilities to social context information.

Although we know that the research in electronic devices and wireless communication is making giant steps, by proposing ever increasing powerful mobile devices and high-bandwidth wireless networks, we think that the real-world realization of context-aware services in large-scale settings is still an extremely complex task. Several factors, including low-level wireless transmissions and bandwidth management, efficient context data storage and processing, and so forth, have to be considered to support quality-based context provisioning in large-scale settings. In addition, the heterogeneity of both mobile devices and involved wireless communications, that exhibit largely different computational power and bandwidth, further complicates the realization of portable CDDIs. All these complexities must be faced by introducing *quality-based* and *resource-aware* CDDI, namely CDDIs capable of granting agreed quality levels while avoid excessive resource consumptions.

In this thesis, we have thoroughly investigated the design and the realization of CDDIs for large-scale mobile systems. We have highlighted different design choices, by considering associated advantages and shortcomings. One of our main claims is that the CDDI has to be able to dynamically adapt to system scalability, while introducing and enforcing quality constraints to enable correct context provisioning on mobile devices. Finally, obtained experimental results have supported the technical soundness of our main claims, while also highlighting further research directions to be investigated.

Considering the main outcomes of this thesis, all the software components of our CDDIs have been implemented in both network simulations and real prototypes. The usage of both these two implementation strategies has allowed to achieve a more complete understanding of context data distribution primitives, since it enables to investigate both the scalability in large-scale mobile systems and the overhead introduced on real-world mobile devices. We recall that all the software components and prototypes developed

during this thesis work can be freely downloaded by the research community; this is a way to foster the building of a research community spanning different research groups all around the world, so as to promote additional and systemic research in this area.

In addition, this thesis work has been realized by mixing together both academic and industrial research. On the one side, the design and the implementation of the RECOWER CDDI has been largely carried out at the PARADISE Research Lab, SITE, University of Ottawa, Canada, under the supervision of Prof. Azzedine Boukerche; on the other one, the design and the implementation of Cloud-based solutions have been investigated during an internship at the IBM Haifa Research Lab, Haifa, Israel, under the supervision of Dr. Ofer Biran and Prof. Danny Raz. Due to those international collaborations, we have established new important connections with external research groups, in order to foster joint collaborations in this research area. In addition, by mixing together academic and industrial research, we have better investigated the possibility of applying our academic and more theoretical research in industrial applications.

The future research directions highlighted by this thesis are manifold. Apart from the more theoretical ones, strictly related with the context data distribution function and discussed in Section 9.2, additional work needs to be done toward the standardization of proper APIs and communication protocols between mobile devices and CDDIs. In fact, the introduction of a common set of communication APIs between CDDI and mobile devices will let service developers focus only on high-level context data requests and usage, while leaving out all the technicalities involved in context data storage, processing, and distribution. At the end, that will build a common ground useful to ease the development of context-aware services, thus fostering their widespread adoption in our society.

In addition, we remark that several industrial efforts and EU funded initiatives, such as IBM Smarter Cities initiative and EU FuturICT project, are currently investigating efficient mechanisms and solutions to build context-aware services in large-scale mobile systems. Such research efforts span the whole software stack of a context-aware system, and present compelling context-aware services that not only sense and reason about the current context situation, but also modify it through proper distributed actuation actions. We think the results of this thesis work can be of extreme interest for all the industries currently entering the area of middleware supports for smart environments, such as the IBM Smarter Cities initiative, since this dissertation largely treated the specific context data distribution function, by introducing main design guidelines and choices. In addition,

from an industrial viewpoint, additional research should be led along proper incentive mechanisms to foster and support the collaborative context data sharing view of proposed CDDIs. Although both ad-hoc wireless communications and context data storage on mobile devices can effectively reduce the data traffic pressure on limited fixed wireless infrastructures, they also result in both higher device overhead and fast battery depletion. Those side-effects can be accepted by mobile users only if counterbalanced by proper incentives, such as discounts for voice calls, free data traffic, extended service offerings, and so on. The design and the realization of such incentive mechanisms are fundamental to prevent and counteract selfish behaviours, with mobile users only care about their own device batteries, thus hindering the collaborative context sharing perspective.

To conclude, we think that the work presented in this dissertation has a general and large applicability to all main classes of context-aware services in future mobile systems. Due to the several outcomes mentioned before, and supported by the publication record obtained from this thesis work, we are very convinced that this thesis can foster future standardization activities in this area and can have an impact and an influence on the design and the realization of CDDIs for next generation mobile systems.

Bibliography

[1] B. N. Schilit, *et al.*, "Context-Aware Computing Applications," in *Workshop on Mobile Computing Systems and Applications (WMCSA'94)*, 1994, pp. 85-90.

[2] A. K. Dey and G. D. Abowd, "Towards a Better Understanding of Context and Context-Awareness," in *Workshop on the What, Who, Where, When, and How of Context-Awareness within CHI'00*, 2000, pp. 1-12.

[3] T. Buchholz, *et al.*, "Quality of Context: What It Is and Why We Need It.," in *Workshop HP OpenView*, 2003, pp. 1-14.

[4] M. Krause and I. Hochstatter, "Challenges in Modelling and Using Quality of Context (QoC)," presented at the International Conference on Mobility Aware Technologies and Applications (MATA'05), 2005.

[5] P. Bellavista, *et al.*, "A Survey of Context Data Distribution for Mobile Ubiquitous Systems " *accepted in ACM Computing Surveys (CSUR),* vol. 45, pp. 1-49, 2013.

[6] G. Chen and D. Kotz, "A Survey of Context-Aware Mobile Computing Research," Dept. of Computer Science, Dartmouth College2000.

[7] A. Manzoor, *et al.*, "On the Evaluation of Quality of Context," presented at the Third European Conference on Smart Sensing and Context, 2008.

[8] K. Cheverst, *et al.*, "Developing a context-aware electronic tourist guide: some issues and experiences," in *SIGCHI conference on Human factors in computing systems (CHI '00)*, 2000, pp. 17-24.

[9] W. G. Griswold. *ActiveCampus Project*. Available: http://activecampus.ucsd.edu/

[10] Q. Jones. *SmartCampus Project*. Available: http://smartcampus.njit.edu/

[11] A. Zimmermann, *et al.*, "An operational definition of context," in *6th International and Interdisciplinary Conference on Modeling and using Context (CONTEXT07)*, 2007, pp. 558-571.

[12] A. Bartolini, *et al.*, "Visual Quality Analysis For Dynamic Backlight Scaling In LCD Systems," in *Design, Automation and Test in Europe (DATE'09)*, 2009, pp. 1428-1433.

[13] S. Ceri, *et al.*, "Model-driven development of context-aware Web applications," *ACM Transactions on Internet Technologies,* vol. 7, pp. 1-33, February 2007 2007.

[14] E. Gustafsson and A. Jonsson, "Always Best Connected," *IEEE Wireless Communications,* vol. 10, pp. 49-55, 2003.

[15] P. Bellavista, *et al.*, "Differentiated Management Strategies for Multi-hop Multi-Path Heterogeneous Connectivity in Mobile Environments," *IEEE Transactions on*

Network and Service Management (IEEE TNSM), vol. 8, pp. 190-204, 2011.

[16] C. Gorgorin*, et al.*, "Adaptive Traffic Lights using Car-to-Car Communication," in *IEEE Vehicular Technology Conference (VTC'07-Spring)*, 2007, pp. 21-25.

[17] U. Lee*, et al.*, "Bio-inspired multi-agent data harvesting in a proactive urban monitoring environment," *Elsevier Ad Hoc Networks,* vol. 7, pp. 725-741, 2009.

[18] J.-M. Kim*, et al.*, " Illuminant Adaptive Color Reproduction Based on Lightness Adaptation and Flare for Mobile Phone," in *IEEE International Conference on Image Processing*, 2006, pp. 1513-1516.

[19] B. Adams*, et al.*, "Sensing and using social context," *ACM Transactions on Multimedia Computing, Communications and Applications,* vol. 5, pp. 1-27, 2008.

[20] P. Eugster*, et al.*, "Middleware Support for Context-Aware Applications," in *Middleware for Network Eccentric and Mobile Applications*, B. Garbinato*, et al.*, Eds., ed: Eds. Springer Press, 2009, pp. 305-322.

[21] A. Gupta*, et al.*, "Automatic identification of informal social groups and places for geo-social recommendations," *International Journal of Mobile Network Design and Innovation (IJMNDI),* vol. 2, pp. 159-171, 2007.

[22] J. Wang*, et al.*, "A sensor-fusion approach for meeting detection," in *Workshop on Context Awareness at the Second International Conference on Mobile Systems, Applications, and Services*, 2004.

[23] A. Manzoor*, et al.*, "Using quality of context to resolve conflicts in context-aware systems," presented at the First International Conference on Quality of Context (QuaCon'09), 2009.

[24] A. Manzoor*, et al.*, "Quality Aware Context Information Aggregation System for Pervasive Environments," in *First International Conference on Advanced Information Networking and Applications Workshops*, 2009, pp. 266-271.

[25] R. Neisse*, et al.*, "Trustworthiness and Quality of Context Information," in *Ninth International Conference for Young Computer Scientists (ICYCS'08)*, 2008, pp. 1925-1931.

[26] C. Bisdikian*, et al.*, "A letter soup for the quality of information in sensor networks," presented at the IEEE International Conference on Pervasive Computing and Communications (PERCOM), 2009.

[27] A. S. Tanenbaum, *Computer Networks*: Prentice Hall, 2002.

[28] A. T. S. Chan and S. N. Chuang, "Mobipads: A reflective middleware for context-aware mobile computing," *IEEE Transactions on Software Engineering,* vol. 29, pp. 1072-1085, 2003.

[29] A. Ranganathan and R. H. Campbell, "A middleware for context-aware agents in ubiquitous computing environments," in *ACM/IFIP/USENIX International Conference on Middleware (Middleware'03)*, 2003, pp. 143-161.

[30] K. Cho, *et al.*, "HiCon: a hierarchical context monitoring and composition framework for next-generation context-aware services," *IEEE Network*, vol. 22, pp. 34-42., 2008.

[31] C. Julien and G.-C. Roman, "EgoSpaces: facilitating rapid development of context-aware mobile applications," *IEEE Transactions on Software Engineering*, vol. 32, pp. 281-298, 2006.

[32] P. Eugster, *et al.*, "Design and Implementation of the Pervaho Middleware for Mobile Context-Aware Applications," presented at the International MCETECH Conference on e-Technologies, 2008.

[33] G. Chen, *et al.*, "Data-centric middleware for context-aware pervasive computing," *Elsevier Pervasive and Mobile Computing*, vol. 4, pp. 216-253, 2008.

[34] T. Hofer, *et al.*, "Context-Awareness on Mobile Devices - the Hydrogen Approach," presented at the 36th Annual Hawaii International Conference on System Sciences, 2003.

[35] O. Riva, *et al.*, "Context-Aware Migratory Services in Ad Hoc Networks," *IEEE Transactions on Mobile Computing*, vol. 6, pp. 1313-1328, 2007.

[36] A. K. Dey and G. D. Abowd, "The Context Toolkit: Aiding the Development of Context-Aware Applications," presented at the Workshop on Software Engineering for Wearable and Pervasive Computing, 2000.

[37] L. Capra, *et al.*, "CARISMA: context-aware reflective middleware system for mobile applications," *IEEE Transactions on Software Engineering*, vol. 29, pp. 929-945, 2003.

[38] L. Pelusi, *et al.* (2006) Opportunistic networking: Data forwarding in disconnected mobile ad hoc networks. *IEEE Communications Magazine*. 134-141.

[39] M. Armbrust, *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley2009.

[40] T.-M. Grønli, *et al.*, "Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments," in *International Conference on PErvasive Technologies Related to Assistive Environments (PETRA'10)*, 2010, pp. 1-8.

[41] (2011). *Global Mobile Data Traffic Forecast Update, 2009-2014*. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html

[42] K. Egan and J. Duvall. (2010). *Mobile data traffic surpasses voice. 2010*. Available: http://www.ericsson.com/thecompany/press/releases/2010/03/1396928

[43] G. Bensinger. (2010). *Wireless Data: The End of All-You-Can-Eat?* Available: http://www.businessweek.com/magazine/content/10_28/b4186034470110.htm

[44] M. Conti and S. Giordano. (2007) Multihop Ad Hoc Networking: The Theory.

IEEE Communications Magazine. 78-86.

[45] M. Conti and S. Giordano. (2007) Multihop Ad Hoc Networking: The Reality. *IEEE Communications Magazine*. 88-95.

[46] J. Whitbeck, *et al.*, "Relieving the wireless infrastructure: When opportunistic networks meet guaranteed delays," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2011, pp. 1-10.

[47] A. Lenk, *et al.*, "What's inside the Cloud? An architectural map of the Cloud landscape," in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009, pp. 23-31.

[48] M. B. S. D. F. Rosenberg, "A Survey on Context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing,* vol. 2, pp. 263-277, 2007.

[49] P. Bellavista, *et al.*, "Context-Aware Middleware for Reliable Multi-hop Multi-path Connectivity," in *6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS '08)*, 2008, pp. 66-78.

[50] *IBM Smarter Planet*. Available: http://www.ibm.com/smarterplanet/us/en/?ca=v_smarterplanet

[51] H. Chang, *et al.*, "Context Life Cycle Management Scheme in Ubiquitous Computing Environments," in *International Conference on Mobile Data Management (MDM'07)*, 2007, pp. 315-319.

[52] P. T. Eugster, *et al.*, "The many facets of publish/subscribe," *ACM Computing Surveys,* vol. 35, pp. 114-131, 2003.

[53] J. Mantyjarvi, *et al.*, "Collaborative context determination to support mobile terminal applications," *IEEE Wireless Communications,* vol. 9, pp. 39- 45, 2002.

[54] T. Hara and S. K. Madria, "Consistency Management Strategies for Data Replication in Mobile Ad Hoc Networks," *IEEE Transactions on Mobile Computing,* vol. 8, pp. 950-967, 2009.

[55] T. Strang and C. L. Popien, "A context modeling survey," in *Workshop on Advanced Context Modelling, Reasoning and Management within UbiComp'04*, 2004, pp. 1-8.

[56] A. Derhab and N. Badache, "Data replication protocols for mobile ad-hoc networks: a survey and taxonomy," *IEEE Communications Surveys & Tutorials,* vol. 11, pp. 35-51, 2009.

[57] P. Padmanabhan, *et al.*, "A survey of data replication techniques for mobile ad hoc network databases," *The VLDB Journal,* vol. 17, pp. 1143-1164, 2008.

[58] C.-Y. Chow, *et al.*, "GroCoca: Group-Based Peer-To-Peer Cooperative Caching In Mobile Environment," *IEEE Journal on Selected Areas in Communications,* vol. 25, pp. 179-191, 2007.

[59] L. Yin and G. Cao, "Supporting Cooperative Caching In Ad Hoc Networks," *IEEE Transactions on Mobile Computing,* vol. 5, pp. 77-89, 2006.

[60] T. Hara, "Effective replica allocation in ad hoc networks for improving data accessibility," in *20th Joint Conference of the IEEE Computer and Communication Societies (INFOCOM'01)*, 2001, pp. 1568–1576.

[61] A. Shaheen and L. Gruenwald, "Group based replication for mobile ad hoc databases (GBRMAD)," University of Oklahoma2010.

[62] M. Hosseini, *et al.*, "A Survey of Application-Layer Multicast Protocols," *IEEE Communications Surveys Tutorials,* vol. 9, pp. 58 -74, 2007.

[63] A. Gaddah and T. Kunz, "A Survey of Middleware Paradigms for Mobile Computing," Dept. of Systems and Computing Engineering, Carleton University2003.

[64] J. Hightower and G. Boriello, "A Survey and Taxonomy of Location Systems for Ubiquitous Computing," *IEEE Computer,* vol. 34, pp. 57-66 2001.

[65] D. A. Chappell and R. Monson-Haefel, *Java Message Service*: O'Reilly Media, 2000.

[66] L. Juszczyk, *et al.*, "Adaptive Query Routing on Distributed Context - The COSINE Framework," presented at the 10th International Conference on Mobile Data Management: Systems, Services and Middleware (MDM '09), 2009.

[67] C. Bolchini, *et al.*, "A data-oriented survey of context models," *SIGMOD Record,* vol. 36, pp. 19-26, 2007.

[68] C. Bettini, *et al.*, "A survey of context modelling and reasoning techniques," *Elsevier Pervasive and Mobile Computing,* vol. 6, pp. 161-180, 2010.

[69] A. Ranganathan, *et al.*, "MiddleWhere: a middleware for location awareness in ubiquitous computing applications," presented at the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware'05), 2004.

[70] U. Hengartner and P. Steenkiste, "Access control to people location information," *ACM Transactions on Information and System Security (TISSEC),* vol. 8, pp. 424-456, 2005.

[71] Q. Jones and S. A. Grandhi. (2005) P3 Systems: Putting the place back into social networks. *IEEE Internet Computing*. 38-46.

[72] R. Friedman, *et al.*, "Gossiping on MANETs: the beauty and the beast," *SIGOPS Operating Systems Review,* vol. 41, pp. 67-74, 2007.

[73] R. Friedman, *et al.*, "Gossip-Based Dissemination," in *Middleware for Network Eccentric and Mobile Applications*, B. Garbinato, *et al.*, Eds., ed: Eds. Springer Press, 2009, pp. 169-190.

[74] A.-M. Kermarrec and M. v. Steen, "Gossiping in distributed systems," *ACM*

SIGOPS Operating Systems Review, vol. 41, pp. 2-7, 2007.

[75] Y. Sasson, *et al.*, "Probabilistic Broadcast for Flooding in Wireless Mobile Ad hoc Networks," in *IEEE Wireless Communications and Networking Conference (WCNC'03)*, 2003, pp. 1124-1130.

[76] V. Drabkin, *et al.*, "RAPID: Reliable Probabilistic Dissemination in Wireless Ad-Hoc Networks," in *26th IEEE Symposium on Reliable Distributed Systems*, 2007, pp. 13-22.

[77] S. Tilak, *et al.*, "Non-uniform Information Dissemination for Sensor Networks," in *11th IEEE Conference on Network Protocols (ICNP'03)*, 2003, pp. 295-304.

[78] A. Cartigny and D. Simplot, "Borden Node Retransmission Based Probabilistic Broadcast Protocols in Ad-Hoc Networks," in *Telecommunication System*, 2003, pp. 189-204.

[79] Z. Haas, *et al.*, "Gossip-based Ad Hoc Routing," in *21st Joint Conference of the IEEE Computer and Communication Societies (INFOCOM'02)*, 2002, pp. 1707-1716.

[80] H. Miranda, *et al.*, "An Algorithm for Dissemination and Retrieval of Information in Wireless Ad Hoc Networks," *John Wiley and Sons, Concurrency and Computation: Practice & Experience,* vol. 21, pp. 889-904, 2009.

[81] N. Roy, *et al.*, "An energy-efficient quality adaptive framework for multi-modal sensor context recognition," presented at the IEEE International Conference on Pervasive Computing and Communications (PERCOM), 2011.

[82] T. Hara, "Quantifying Impact of Mobility on Data Availability in Mobile Ad Hoc Networks," *IEEE Transactions on Mobile Computing,* vol. 9, pp. 241-258, 2010.

[83] A. Senart, *et al.*, "Vehicular Networks and Applications," in *Middleware for Network Eccentric and Mobile Applications*, B. Garbinato, *et al.*, Eds., ed: Eds. Springer Press, 2009, pp. 369-382.

[84] H. Hartenstein and K. Laberteaux, "Introduction," in *VANET Vehicular Applications and Inter-Networking Technologies*, ed: John Wiley & Sons, 2010.

[85] Z. Zhang, "Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges," *IEEE Communications Surveys & Tutorials,* vol. 8, pp. 24-37, 2006.

[86] K. Fall, "A Delay-tolerant Network Architecture for Challenged Internets," in *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*, 2003, pp. 27-34.

[87] K. Pan, *et al.*, "Implementation of Data Distribution Management services in a Service Oriented HLA RTI," presented at the Proceedings of the 2009 Simulation Conference (WSC), 2009.

[88] V. Jacobson. *Content-Centric Networking Resources*. Available:

http://www.ccnx.org/documentation/content-centric-networking-resources-2/

[89] M. Varvello, *et al.*, "On The Design Of Content-Centric MANETs," presented at the Eighth International Conference on Wireless On-Demand Network Systems And Services (WONS).

[90] S. Y. Oh, *et al.*, "Content Centric Networking in Tactical And Emergency MANETs," presented at the IFIP Wireless Days, 2010.

[91] OMG. *Data Distribution Service for Real-Time Systems Specification*. Available: http://www.omg.org/docs/formal/04-12-02.pdf

[92] S. P. Mahambre, *et al.* (2007) A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Computing*. 35-44.

[93] G. Cugola and E. D. Nitto, "Using a Publish/Subscribe Middleware to Support Mobile Computing," in *Workshop on Middleware for Mobile Computing (MMC'01) within Middleware'01*, 2001, pp. 1-5.

[94] G. Cugola, *et al.*, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827-850, 2001.

[95] G. Muhl, *et al.* (2004) Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Computing*. 46- 53.

[96] P. Sutton, *et al.*, "Supporting Disconnectedness-Transparent Information Delivery for Mobile and Invisible Computing," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, 2001, pp. 277-285.

[97] N. Aschenbruck, *et al.*, "Modelling mobility in disaster area scenarios," in *10th ACM Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MsWIM'07)*, 2007, pp. 4-12.

[98] T. Catarci, *et al.* (2008) Pervasive Software Environments for Supporting Disaster Responses. *IEEE Internet Computing*. 26-37.

[99] Q. Jones, *et al.*, "People-to-People-to-Geographical-Places: The P3 Framework for Location-Based Community Systems," *Comput. Supported Coop. Work*, vol. 13, pp. 249-282, 2004.

[100] R. Buyya, *et al.*, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Elsevier Future Generation Computer Systems*, vol. 25, pp. 599-616, 2009.

[101] M. Fanelli. (2010). *RECOWER CDDI*. Available: http://lia.deis.unibo.it/Research/RECOWER/

[102] "Middleware for Network Eccentric and Mobile Applications," B. Garbinato, *et al.*, Eds., ed, 2009, p. 454.

[103] J.-H. Cho, *et al.*, "A Survey on Trust Management for Mobile Ad Hoc Networks,"

IEEE Communications Surveys & Tutorials, vol. 13, pp. 562-583, 2011.

[104] A. B. McDonald and T. F. Znati, "A mobility-based framework for adaptive clustering in wireless ad hoc networks," *IEEE Journal on Selected Areas in Communications,* vol. 17, pp. 1466-1487, 1999.

[105] Y.-C. Tseng, *et al.*, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," *Springer Wireless Network,* vol. 8, pp. 153-167, 2002.

[106] M. Fanelli, *et al.*, "QoC-based Context Data Caching for Disaster Area Scenarios," presented at the IEEE International Conference on Communications (ICC '11), Kyoto, Japan, 2011.

[107] M. Fanelli, *et al.*, "Self-Adaptive and Time-Constrained Data Distribution Paths for Emergency Response Scenarios," presented at the 8th ACM Symposium on Mobility Management and Wireless Access (MOBIWAC'10), Bodrum, Turkey, 2010.

[108] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM,* vol. 13, pp. 422-426, 1970.

[109] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics,* vol. 1, pp. 485-509, 2005.

[110] M. Y. S. Uddin, *et al.*, "A Post-Disaster Mobility Model For Delay Tolerant Networking," in *2009 IEEE Winter Simulation Conference*, 2009, pp. 2785-2796.

[111] W. Kang, *et al.*, "PRIDE: A Data Abstraction Layer for Large-Scale 2-tier Sensor Networks," in *6th IEEE Communications Society Conference on Sensor, Mesh and Ad-hoc Communications and Networks (SECON 2009)*, 2009, pp. 1-9.

[112] M. Fanelli. (2010). *SALES CDDI*. Available: http://lia.deis.unibo.it/Research/SALES/

[113] A. M.F.Caetano, *et al.*, "A collaborative cache approach for mobile ad hoc networks," presented at the 14th IEEE Symposium on Computers and Communications (ISCC), 2009.

[114] Y.-H. Wang, *et al.*, "A distributed data caching framework for mobile ad hoc networks," presented at the International Conference On Communications And Mobile Computing, 2006.

[115] N. Chand, *et al.*, "Efficient Cooperative Caching in Ad Hoc Networks," presented at the First International Conference on Communication System Software and Middleware (COMSWARE), 2006.

[116] (2011). *Pearson product-moment correlation coefficient*. Available: http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

[117] A. Johnsson, *et al.*, "An Analysis of Active End-to-end Bandwidth Measurements in Wireless Networks," presented at the 4th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, 2006.

[118] E. Kayacan, *et al.*, "Grey system theory-based models in time series prediction," *Elsevier Expert Systems with Applications,* vol. 37, pp. 1784-1789, 2010.

[119] R. Meier, *Professional Android 2 Application Development*: John Wiley and Sons, 2010.

[120] (2011). *JSR-82 Bluetooth API*. Available: http://java.sun.com/javame/reference/apis/jsr082/

[121] S. Zammit and D. Catania, "Video Streaming over Bluetooth," presented at the WICT'08., 2008.

[122] C. Hyser, *et al.*, "Autonomic Virtual Machine Placement in the Data Center," HPL-2007-189, 2007.

[123] X. Meng, *et al.*, "Improving the scalability of data center networks with traffic-aware virtual machine placement," presented at the 29th conference on Information Communications (INFOCOM'10), 2010.

[124] A. Corradi, *et al.*, "Adaptive Context Data Distribution with Guaranteed Quality for Mobile Environments," presented at the IEEE Int. Symp. on Wireless Pervasive Computing (ISWPC'10), 2010.

[125] B. Han, *et al.*, "Cellular traffic offloading through opportunistic communications: a case study " presented at the 5th ACM workshop on Challenged networks (CHANTS '10), 2010.

[126] M. Musolesi and C. Mascolo, "CAR: Context-aware Adaptive Routing for Delay Tolerant Mobile Networks," *IEEE Transactions on Mobile Computing,* vol. 8, pp. 246-260, 2009.

[127] C. Boldrini, *et al.*, "Exploiting users' social relations to forward data in opportunistic networks: The HiBOp solution," *Elsevier Pervasive and Mobile Computing,* vol. 4, pp. 633-657, 2008.

[128] Amazon. (2012). *Amazon Elastic Compute Cloud (Amazon EC2)*. Available: http://aws.amazon.com/ec2/

[129] M. Wang, *et al.*, "Consolidating Virtual Machines with Dynamic Bandwidth Demand in Data Centers," presented at the IEEE INFOCOM 2011 MINI-CONFERENCE, 2011.

[130] M. Korupolu, *et al.*, "Coupled Placement in Modern Data Centers," presented at the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2009.

[131] A.Singh, *et al.*, "Server-storage virtualization: integration and load balancing in data centers," presented at the ACM/IEEE Conference on Supercomputing (SC '08), 2008.

[132] Y. Toyoda, "A simplified algorithm for obtaining approximate solutions to zero-one programming problems," *Management Science,* vol. 21, pp. 1417-1427, 1975.

[133] A. Greenberg, *et al.*, "VL2: a scalable and flexible data center network," presented at the ACM SIGCOMM 2009 conference on Data communication (SIGCOMM '09), 2009.

[134] M. Al-Fares, *et al.*, "A scalable, commodity data center network architecture," presented at the ACM SIGCOMM 2008 conference on Data communication (SIGCOMM '08), 2008.

[135] C. Guo, *et al.*, "BCube: a high performance, server-centric network architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 63-74, 2009.

[136] *Cisco data center infrastructure 2.5.*

[137] A. J. Mashhadi, *et al.*, "Habit: Leveraging Human Mobility and Social Network for Efficient Content Dissemination in MANETs," presented at the 10th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'09), 2009.

Publications

- **A Survey of Context Data Distribution for Mobile Ubiquitous Systems**, P. Bellavista, A. Corradi, M. Fanelli, L. Foschini, Accepted in ACM Computing Surveys (CSUR), ACM Press, expected to appear in Vol. 45, No. 1, Mar 2013, pages 1-49.

- **A Stable Network-Aware VM Placement for Cloud Systems,** O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, E. Silvera, Proceedings of the IEEE CCGrid'12 conference, Ottawa, Canada, May, 2012, IEEE Computer Society Press.

- **Context Data Distribution in Mobile Systems: a Case Study on Android-based Phones,** A. Corradi, M. Fanelli, L. Foschini, M. Cinque, Proceedings of the IEEE International Conference on Communications (ICC'12), Ottawa, Canada, June, 2012, IEEE Computer Society Press.

- **Resource-Awareness in Context Data Distribution for Mobile Environments**, M. Fanelli, L. Foschini, A. Corradi, A. Boukerche, Proceedings of the IEEE Global Communications Conference (GLOBECOM'11), Houston, Texas, USA, Dec. 5-9, 2011, IEEE Computer Society Press.

- **QoC-based Context Data Caching for Disaster Area Scenarios,** M. Fanelli, L. Foschini, A. Corradi, A. Boukerche, Proceedings of the IEEE International Conference on Communications (ICC'11), Kyoto, Japan, July, 2011, IEEE Computer Society Press.

- **Increasing Cloud Power Efficiency through Consolidation Techniques**, A. Corradi, M. Fanelli, L. Foschini, Proceedings of the IEEE Workshop on Management of Cloud Systems (MoCS 2011), Kerkyra (Corfu), Greece, June 28, 2011, IEEE Computer Society Press.

- **Counteracting wireless congestion in data distribution with adaptive batching techniques**, M. Fanelli, L. Foschini, A. Corradi, A. Boukerche, Proceedings of the IEEE Global Communications Conference (GLOBECOM'10), Miami, Florida, USA, Dec. 6-10, 2010, IEEE Computer Society Press.

- **Self-Adaptive and Time-Constrained Data Distribution Paths for Emergency Response Scenarios**, M. Fanelli, L. Foschini, A. Corradi, A. Boukerche, Proceedings of the 8th ACM Symposium on Mobility Management and Wireless Access (MOBIWAC'10), Bodrum, Turkey, Oct. 17-21, 2010, ACM Press.

- **Towards Efficient and Reliable Context Data Distribution in Disaster Area Scenarios**, M. Fanelli, L. Foschini, A. Corradi, A. Boukerche, Short paper in the Proceedings of the 35th IEEE Conference on Local Computer Networks (LCN'10), Denver, Colorado, USA, Oct. 11-14, 2010, IEEE Computer Society Press.

- **Adaptive Context Data Distribution with Guaranteed Quality for Mobile Environments**, A. Corradi, M. Fanelli, L. Foschini, Proceedings of the IEEE International Symposium on Wireless Pervasive Computing (ISWPC'10), Modena, Italy, May 5-7, 2010, IEEE Computer Society Press.

- **Towards Adaptive and Scalable Context-Aware Middleware**, A. Corradi, M. Fanelli, L. Foschini, Invited article in International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS), Vol. 1, 2010, IGI-Global Press.

- **Implementing a Scalable Context-Aware Middleware**, A. Corradi, M. Fanelli, L. Foschini, Proceedings of the 14th IEEE International Symposium on Computers and Communications (ISCC'09), Sousse, Tunisia, Jul. 5-8, 2009, IEEE Computer Society Press.

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor Prof. Antonio Corradi and Dr. Luca Foschini for their precious guide during my thesis years. They have followed my work with constant patience, and by providing continuous encouragements. Also, a special thank goes to Prof. Paolo Bellavista for the several advices and discussions, and to Prof. Eugenio Faldella for his moral and human support.

I would like to thank Prof. Azzedine Boukerche for his technical guidance during my abroad stay in Canada; he taught me a lot of important things on the research world. Special thanks go to both Dr. Ofer Biran and Prof. Danny Raz, for their fundamental technical guide during my stay in Israel; their continuous advices and support helped me in getting the best out of my internship at IBM Haifa Research Lab. Finally, I want to truly thank all the external reviewers of my PhD thesis, in particular, Prof. Azzedine Boukerche, Prof. Patrick Eugster, and Prof. Archan Misra, for their useful and important advices about my work.

A special thank goes to my friends Giuseppe, Primiano, Andrea, Luca, and Alessio, who shared with me hard and fun periods during my PhD; I think I will never forget all the nerdy discussions of our lunches/dinners. Also, I want to thank my Canadian friends, especially Michael and Julie, Robson and Priscila, Richard, Daniel and Fernanda, Kent, Cristiano, Leandro, and Haifa, for all the fun we had in Ottawa.

Finally, I would like to thank all my family, for their continuous human support and for their great faith in me and my decisions: thank you Mum, Dad, Francesca, and Andrea, for being there every time I needed.