UNIVERSITÀ DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXVIII Ciclo

An Autonomic Framework for Mobile Cloud Computing

Coordinatore:

Chiar.mo Prof. Marco Locatelli

Tutor:

Chiar.mo Prof. Francesco Zanichelli

Dottorando: *Alessandro Grazioli*

Gennaio 2016

To my family,

thanks for everything you give me everyday

Summary

List of Figures

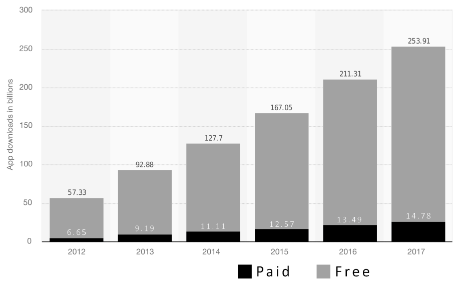
List of Tables

Introduction

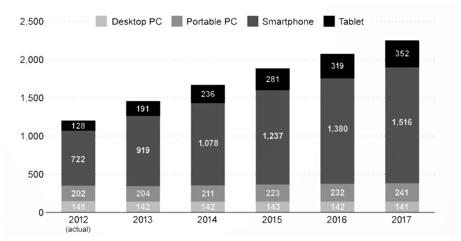
In recent years, the market of mobile devices – smartphones, tablets, PDAs, Mobile Internet Devices (MIDs), portable media players (PMPs), netbooks, etc. – has been relentlessly growing. Nowadays, mobile applications represent an important segment of the global economy. The ever increasing capabilities of such devices make them attractive to users, in that they provide a huge amount of applications and services to be experienced in mobility, with reference to information, business and entertainment domains. Fig. 1[1]illustrates mobile market trends in the last years. Fig. 1 (a) shows the estimated number of paid and free apps downloads worldwide from 2012 to 2017. Fig. 1 (b) shows the estimated shipments trend for traditional PCs and mobile devices until 2017. It can be seen that, while the shipments numbers for both desktop and portable devices remain approximately constant, both smartphones and tablet sales consistently grow. Thus, mobile devices are nowadays becoming the commonest computing platform of choice. One consequence of such a trend is that, as Fig. 1 (c) presents, the mobile data traffic originating from smartphones and tablets will consistently grow as well. Finally, Fig. 1 (d) illustrates the market fragmentation as several vendors compete to acquire the highest position, being Samsung, Microsoft and Apple the most successful ones.

Such trends have been considered for a long time by the *Mobile* and *Ubiquitous Computing* research areas. Specifically, the latter term is generally used to describe techniques and technologies that enable people to access computing services anyplace, anytime and anywhere.

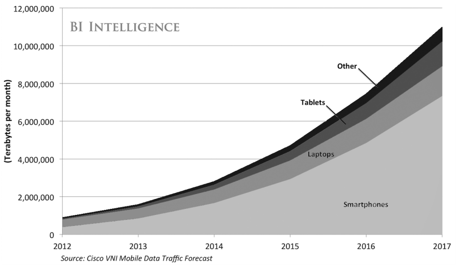
[1]Data from `http://www.statista.com/`.



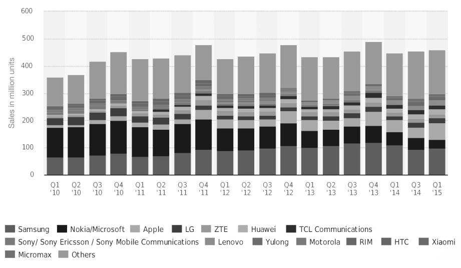
(a) Estimated number of downloaded free and paid apps from 2012 to 2017 (in billion units), by year.



(b) Estimated device shipments by form factor worldwide until 2017 (in million units), by year.



(c) Global mobile data traffic forecast by form factor from 2012 to 2017 (in Terabytes), by month.



(d) Vendors' sales of mobile phone sales to end users worldwide from 2010 to 2015 (in million units), by quarter.

Figure 1: Mobile market trends.

Mobile Computing encompasses all the aspects concerning mobile hardware, software and communication. Regarding the hardware, device capabilities constantly improve due to market-driven research efforts. Nowadays, smartphones and tablets are provided with multi-core CPUs, GPUs, Gigabytes of RAM, several network interfaces and a relevant number of sensors such as camera, accelerometer, GPS, microphone and compass. Such improvements and the ubiquitous availability of network connectivity are changing the applications addressed by mobile devices that are becoming sources, aggregators and processors of heterogeneous information.

Mobile Cloud Computing

If compared to desktop PCs and servers, mobile devices are characterized by limits concerning performance and battery, when they are required to execute intensive tasks. Cloud Computing (CC) is a way to extend the capabilities of such devices through the temporary and on-demand ubiquitous access to computing and storage resources provided by remote cloud infrastructures. Augmented devices are thus able to perform intensive tasks, extensive computation and store huge amount of data, beyond their intrinsic capabilities.

Mobile cloud computing (MCC) is an emerging paradigm for transparent elastic augmentation of mobile devices capabilities aiming at increasing the range of resource-intensive tasks supported by mobile devices, while preserving and extending their resources. Its main concerns regard the augmentation of energy efficiency, storage capabilities, processing power and data safety, to improve the experience of mobile users. The design of MCC systems is a challenging task, in that both the mobile device and the Cloud have to find energy-time tradeoffs and the choices on one side affect the performance of the other side. The thorough comprehension of critical factors, such as the current state of local and remote resources and the available bandwidth, which strongly influence the augmentation process, is of capital importance.

The Cloud can be considered as a distributed system made of a cluster of computing nodes accessible as a unified resource based on a Service Level Agreement (SLA) [1]. The SLA is a formal contract negotiated in advance between service providers and consumers to assess a level of quality against a fee. The goal of enabling the seamless provision, acquisition and release of shared computing resources through the ubiquitous and on-demand network access, pushes researchers to investigate new solutions for minimizing required management efforts and increasing performance.

Applications that benefit from MCC principles can be organized in the following categories [2].

- *Computing-intensive software*, such as natural language processing, augmented reality and video, image and speech recognition;
- *Data-intensive programs*, such as enterprise applications;

- *Communication-intensive applications*, such as online video streaming.

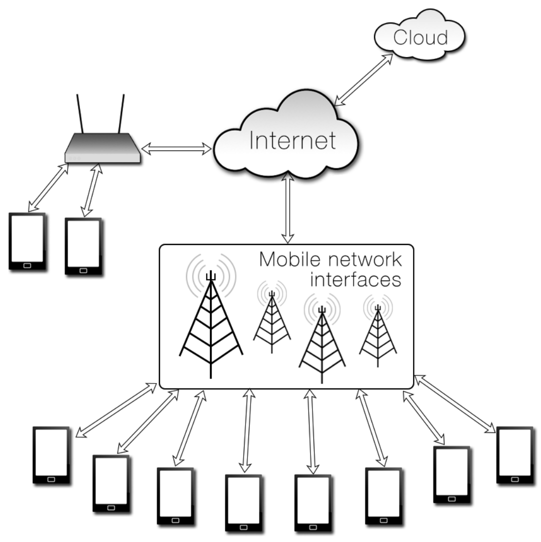
General model and concept of MCC

In the last decade, the continuous growth in expectations regarding mobile applications has given rise to the need to enhance the computing power of mobile devices. Also, the concepts of load sharing and remote execution has risen with the aim of achieving such an enhancement by migrating computational-intensive tasks to *surrogates* (*i.e.*, high performance computing devices). Such an approach led to interesting results, but its actual usefulness is hindered by a number of factors such as the fact that servers can abruptly become unavailable and access and tamper offloaded data [3]. The emergence of the cloud computing paradigm allowed to overcome most of surrogates issues. The vision of the cloud as a model for the ubiquitous and on-demand network access to a virtually unlimited amount of shared and configurable resources lured researches' attention and cloud computing is currently the most promising approach for mobile devices enhancement.

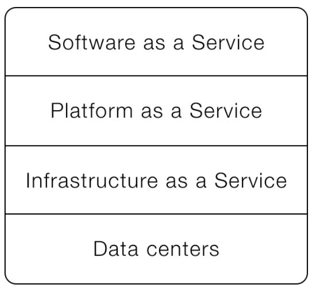
MCC leverages on different cloud-based computing resources, the most important of which are *distant clouds* and near *mobile devices*. Distant clouds, such as Amazon EC2[2], are extremely powerful infrastructures, located far from users, that provide virtually unlimited resources whose access is characterized by high WAN latency. On the other hand, near mobile devices can constitute a cluster and share their resources to provide cloud-like services characterized by lower performance and latency when compared to distant clouds.

Fig. 2 (a) presents the general architecture of a MCC system. User devices are connected to mobile networks via base stations or satellites that establish and manage the interfaces. Mobile operators can provide services to users such as authentication, authorization, and accounting based on subscribers' data. Transmitted data are then forwarded to the cloud through the internet. The cloud then processes requests to provide services [4]. Cloud services are generally classified as a layered stack, as illustrated by fig. 2 (b).

[2]Amazon Elastic Compute Cloud: http://aws.amazon.com/it/ec2/



(a) General structure.



(b) Cloud services.

Figure 2: Mobile Cloud Computing.

Resource heterogeneity is useful to augment the number of supported applications, however MCC applications present a number of critical questions that need to be answered before implementing them as MCC systems. For example, when is MCC actually beneficial and when not? What kind of resources are needed by the application? Is data confidentiality needed? Is the access to local resources or sensors required? What are users preferences?

MCC approaches

During the years, several approaches aiming at augmenting mobile devices capabilities have been proposed and studied. Such approaches can be both hardware and software. The hardware approach concerns the technological evolution of physical components such as CPU, memory, battery, communication infrastructures, sensors and storage. The main software approaches include load sharing, remote execution and cyber foraging. Such approaches imply the migration of the whole task. Operations such as the identification of data and code to be migrated, decisions regarding whether to offload or not and the actual offloading are intensive tasks and consume mobile device resources. Storing and re-using code on nodes ubiquitously accessible by every device – such as a cloud infrastructure – reduces such a problem. Abolfazli *et al.* [2] coined the *Cloud-based Augmentation for mobile devices* (CMA) term to include all approaches aiming at extending mobile devices capabilities following such a perspective.

In the late 90s Othman and Hailes [5] were among the first to propose a software approach aiming at preserving mobile resources by adopting the distributed systems' load balancing approach. *Load Sharing* consists in sharing computational load among the nodes of a distributed system, without attempting to equalize it. The goal is to improve the use of resources by making sure that no node is idle. A general load sharing algorithm consists of two policies. A *transfer policy* decides when jobs should be transferred based on metrics such as the job size and available bandwidth. A *location policy* decides jobs' destinations (*i.e.*, to which host a job will be transfered). Such a process can choose a random node or decide based on workload information that can be obtained by probing nodes or by periodically collecting information. In the architecture, a mobile device sends the job to a base station responsible for finding a suitable node, forwards the job to it, receive results and send them back. Receiving hosts are servers and desktop PCs, so no mobility support is provided. Also, state migration is not available.

Remote Execution is another approach emerged in the 90s aiming at augmenting mobile devices capabilities through execution and data storage on surrogates. Rudenko *et al.* [6] investigated such an approach mainly focusing on portable computers power saving. Such devices are in fact also characterized by power management issues that hinder their utility in that battery life is not always sufficient. Tasks whose nature does not require local execution can be offloaded. The authors demonstrated that if the power needed to remotely send a task, waiting while idle and receiving results is lower than the one required by local execution, migration can save a remarkable amount battery life depending on the size of such a task. A drawback of Rudenko's approach is the static partitioning. In fact, the latter is a difficult task in that it heavily depends on the application and platform. As mobile devices evolve rapidly, the optimal partitioning changes over time. Also, the environment affects connectivity and bandwidth. Balan *et al.* [7] investigated tactics aiming at improving applications' partitioning into local and remote components. Flinn *et al.* [8] designed and implemented a self-tuning remote execution system for pervasive computing which addresses the complexities of pervasive computing to improve both performance and battery life. Although such efforts showed positive results, several issues hinder such an approach. Firstly, offloading decisions are a difficult task since many factors have to be considered, such as performance differences between client and server nodes and the available bandwidth. Secondly, servers are characterized by security and reliability issues in that data can be accessed without authorization, tampered and, in the absence of a SLA, services are not guaranteed [3].

Cyber Foraging is an approach proposed by Satyanarayanan [9] for which wireless mobile devices' computing resources can augment through the exploitation of wired hardware infrastructures. A mobile device senses the environment, looks for potential surrogates and negotiates their use. The author envisions a scenario where mobile devices and surrogates communicate through short-range wireless peer-to-peer (P2P) technologies. The surrogate can act as a gateway to the internet and, in case of intensive computation, the device can send tasks to the surrogate for execution.

Kovachev and Klamma [10] classified MCC approaches found in the literature in three categories.

- *Augmented execution* addresses computation, memory and battery limitations for mobile devices through execution offloading to a computational infrastructure (cloud). The latter runs a virtual machine (VM) -based cloned replica of the smartphone's software. In this application model, the mobile device hosts demanding applications. However, some or all tasks can be offloaded to a cloud where a cloned system image of the device is running. Results are then reintegrated upon completion. Such an approach requires loosely synchronized replicas of the device which are instantiated based on policies that take into account the performance-cost tradeoff.

- *Elastic partitioned applications* improve their performance by delegating part of the computation to a resource-rich cloud infrastructure. A fundamental aspect of such an approach is elasticity, that is the ability to acquire and release resources on demand. The offloading granularity can range from complete software modules to single methods, and an underlying runtime management platform hides most of complexity concerning the deployment, execution and maintenance of such applications.

- An *ad-hoc mobile cloud* is a system made of a group of connected mobile devices which acts as a cloud. Such devices expose their computing resources to other mobile devices. This approach becomes more interesting in situations with no or weak connections to the Internet and large cloud providers. Offloading to nearby mobile devices save monetary cost in that data charging is avoided. Moreover, it fosters the creation of computing communities in which users can collaboratively execute shared tasks.

The aforementioned approaches can be placed on a planar space to compare the cloud involvement and offloading grain, as presented by Fig. 3. The augmented execution approach is characterized by high cloud usage and coarse-grained offloading, while the other approaches present fine-grained offloading and a different degree of cloud involvement.

Issues and disadvantages of MCC

MCC can augment devices' capabilities to make them able to perform intensive tasks, extensive computation and store amounts of data beyond their intrinsic capabilities. However, such a technology is characterized by a number of issues, such as the increased complexity in both design and implementation, and the risk of unauthorized access to data and resources stored on the cloud. Also, MCC systems strongly depend on network infrastructures. Thus, a high performance, robust and reliable communication among cloud infrastructures and mobile devices is required. Currently,

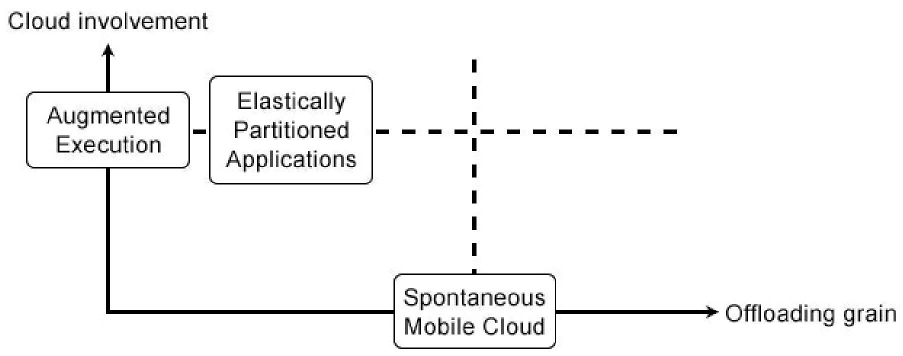


Figure 3: Cloud involvement vs offloading grain for MCC approaches.

network throughput and ubiquitous connectivity present deficiencies that prevent the wide spreading of such an approach, due to possible loss of service quality and performance degradation. Another difficult task is the estimation of energy consumption and processing power required to perform tasks that can be offloaded. Such data are used by the decision engine, a fundamental part of MCC systems. Moreover, other challenges such as the execution state migration, the efficient allocation of cloud resources, trust and security, exist. Finally, cloud access is usually paid and not every user is willing to afford such a price, depending on the application.

Thesis contribution

This Ph.D. thesis[3] focuses on the problem of defining a conceptual model for MCC systems. The lack of an agreed upon model is in fact one of the main hindrances to the full realization of MCC. Also, we investigate the impact of autonomic policies on MCC in that, as multiple offloading approaches are possible [10] depending on the task and context, autonomic computing techniques appear promising to increase the robustness and flexibility of MCC applications [11].

[3]This Ph.D fellowship has been partly funded by a grant from Spinner Consortium, titled "Mobilità Urbana Cooperativa in una Città Amichevole", and partly by the Dipartimento di Ingegneria dell'Informazione of Parma University.

The design of MCC systems is a challenging task, because both the mobile device and the Cloud have to find energy-time tradeoffs and the choices on one side affect the performance of the other side. Through the analysis of MCC literature, we noticed that existing models focus on mobile devices, considering the Cloud as a system with unlimited resources. Therefore, we study MCC analytical aspects through the definition of a modeling and testing framework for cloud-based systems. Such a framework encompasses both cloud and devices components and allows us to better characterize the cloud behavior. We also implement autonomic policies and analyze the influence they have on MCC applications. Such policies aim at automatically determine energy-time tradeoffs and achieve better global performance through autoscaling strategies.

We also study the integration of peer-to-peer (P2P) and cloud systems to leverage the advantages of both approaches. The superior availability of the Cloud makes it a more appealing environment for firms when compared to the best effort philosophy of P2P. However, the Cloud alone may be not sufficient to achieve cost-effective and efficient large scale distributed collaborative environments.

The main contributions of this thesis can be summarized as follows.

- The definition of a novel MCC framework based on Networked Autonomic Machine (NAM) [12], a general-purpose conceptual tool which describes distributed autonomic systems and is suitable for MCC systems as well, as it supports code, execution state and data mobility concepts.
- The extension of the Java implementation of the framework, supporting the deployment of distributed systems whose nodes can migrate computation to each other. Migration only happens when nodes autonomically decide it is beneficial.
- The definition of a modeling and testing framework for the design and analysis of cloud-based systems, which allows to study how autonomic policies impact on MCC applications. To this purpose, we developed a discrete event simulator for the evaluation of MCC systems.

- The study of how the integration of P2P principles impacts on MCC systems, to improve the applications' cost-effectiveness while maintaining satisfactory performance.

Thesis outline

The thesis is organized as follows.

- Chapter 1 presents MCC-related computational paradigms such as Cloud, P2P and Autonomic computing. The features of such paradigms are analyzed to better understand the choices we made in designing the NAM-based MCC framework.

- Chapter 2 describes and analyzes the literature of systems related to MCC and autonomic computing. Among several applications and research projects we selected the most relevant and innovative approaches to illustrate current state of the art and motivate the solutions proposed in this thesis.

- Chapter 3 formally describes the NAM-based framework and presents its components and mobility actions allowing for code and state migration.

- Chapter 4 presents the Java implementation of the MCC framework.

- Chapter 5 presents our studies on how autonomic policies and the integration of P2P principles impact on MCC systems. Also, a MCC modeling and testing framework is presented together with application examples.

- Chapter 6 concludes this thesis and outlines further work.

Chapter 1

Background

This chapter introduces background computational paradigms and presents features, benefits and issues of each, to better understand the choices we made in designing the NAM-based MCC framework.

1.1 Cloud Computing

In recent years, commodity services accessed by users based on their needs, regardless of where such services reside, are becoming commonplace. Cloud computing is a computational paradigm that encompasses the design of computing systems and application development, based on the concept of *dynamic provisioning* applied to services, computing and storage capabilities made available through the Internet on a pay-per-use basis [13]. The paradigm refers to the whole computing stack made of high-level applications delivered as services and the underlying hardware [14].

Consumers only pay when services are needed and do not have to face configuration and maintenance of complex infrastructures, as the computing system's stack is rendered into a collection of services which can be composed together to deploy the required system, with virtually no maintenance costs. The Cloud computing paradigm is supported by high performance data centers which leverage virtualization technologies to provide *everything-as-a-service (XaaS)* where the *X* can represent software *(SaaS)*, computing resources *(PaaS)* and infrastructures *(IaaS)* as illustrated by Fig. 1.1.

SaaS is a model in which software is centrally hosted and licensed on a subscription basis. Services are typically accessed by users via a web browser. Such a model has been adopted by a remarkable number of applications, such as customer relationship management *(CRM)*, office applications, enterprise resource planning *(ERP)*, video and photo editing, DBMS, CAD, messaging, development tools, accounting, collaborative environments *(CE)*, management information systems *(MIS)*, social networking and content management *(CM)* [13]. SaaS architectures generally follow a multi-tenancy model, for which a single configuration is used by multiple users. Scalability is provided through horizontal scaling (*i.e.*, the application is deployed on several physical nodes). Also, SaaS applications support customization, thus allowing users to specify their own preferences and alter the look-and-feel and functionalities. Since services are centrally hosted and cannot access enterprises' internal systems, they are mainly based on WAN protocols such as HTTP, REST and SOAP. Examples of SaaS providers are Windows Azure, Apple iCloud, Amazon Web Services and Google Apps.

PaaS is a model which provides scalable and elastic platforms and runtime environments allowing users to execute applications. A middleware takes care of every aspect regarding the creation of the abstract environment on which applications get deployed and executed [13]. PaaS can be used to develop and integrate services such as database integration, security management and team collaboration with reduced costs and without the burden of hardware and software configuration, optimization and maintenance. As for *SaaS*, the architecture follows a multi-tenancy model and is based on WAN protocols.

IaaS is a model to provide hardware, storage and networking capabilities on demand. Virtual hardware provides computing in the form of virtual machines which are set up on demand over provider's infrastructure [13].

Web 2.0 technologies play a central role through service orientation – which provides abstraction – and virtualization – which confers customization and flexibility.

Many actors (*e.g.*, enterprises, research centers, education institutions, governments, private users) have started using services following such a paradigm, thus pushing research efforts which enriched the set of provided services and reduced prices.

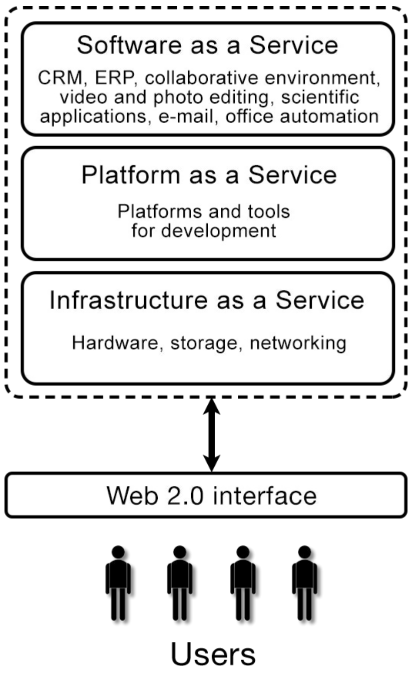

Software as a Service
CRM, ERP, collaborative environment, video and photo editing, scientific applications, e-mail, office automation
Platform as a Service
Platforms and tools for development
Infrastructure as a Service
Hardware, storage, networking
Web 2.0 interface
Users

Figure 1.1: Cloud computing general model.

1.1.1 Cloudlets

High latencies represent an obstacle to performance augmentation of mobile devices in MCC systems, in that they reduce usability. Humans are sensitive to delay and jitter, thus immersive tasks such as multimedia streaming, online gaming and augmented reality, may become unpleasant. To obtain the benefits of cloud computing without the limits of WAN latency, augmentation can happen through *cloudlets* [15]. A cloudlet is a proximate cloud composed of a set of resource-rich devices usually connected through a low-latency high-bandwidth network. A cloudlet provides its resources on a local network, thus minimizing transport overhead due to one-hop connections with mobile clients. Also, as presented by Fig. 1.2, cloudlets can be connected to a distant cloud to leverage its computing power, if the cloudlet's performance is not enough for remarkably demanding applications. The mobile device acts as a thin client and most of the computation happens on the cloudlet. If no cloudlet is available on the local network, the device can switch to the remote cloud or perform tasks locally. Cloudlets are decentralized, self-managing infrastructure components whose maintenance only requires power, a network connection and access control for setup. Such small data centers are thus easy to deploy. Internally, a cloudlet is a cluster of multicore computers, with gigabit internal connectivity and a high-bandwidth wireless network interface.

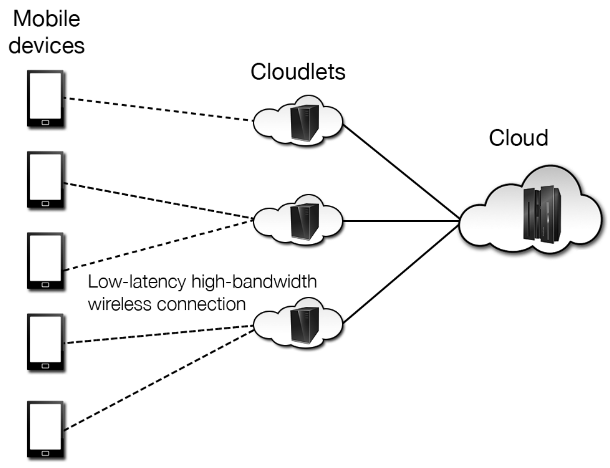


Figure 1.2: Architecture of a cloudlet-based system.

1.1.2 Cloud benefits

Cloud computing brings several benefits to both consumers and providers [13]. One of the main benefits is represented by reduced maintenance and operational costs, as IT assets become utilities accessed only when necessary. Thus, capital costs (*i.e.,* costs associated with assets that need to be paid in advance to start an activity) get substantially reduced. Another important benefit is the increased flexibility in the definition of software systems. In fact, the composition of the latter happens with no constraints on capital costs. Also, the ease of scalability across the entire computing stack is granted by the fact that cloud infrastructures are provided with a huge amount of resources. Another benefit for end users is the ubiquitous access to data and computation through web-based, platform-independent interfaces. Finally, the aggregation of existing services through the on-demand provisioning of cloud resources fosters the birth of new services.

1.1.3 Cloud challenges

As for the benefits, cloud computing is also characterized by a number of issues which affect both consumers and providers [13]. Aspects such as the deployment and configuration of cloud systems come along with challenges related to the dynamic provisioning of resources. Privacy and confidentiality issues emerge since consumers entrust private data to a 3rd party infrastructure. Providers are required to prove compliance to security standards. However, a malicious provider can easily obtain data stored on VMs.

1.2 Peer-to-peer Computing

Peer-to-peer (P2P) is a model for which network nodes are able to directly exchange resources and provide services between each other without the need for centralized servers [16]. P2P systems are also able to aggregate resources and data from nodes to perform shared tasks. Tipically, each P2P node hosts metadata describing shared resources and known peers, thus allowing and facilitating the search on the network. Resource discovery queries are routed around the network through such metadata to nodes hosting the required resources. Also, resources can be stored on a single peer or on a set of peers, each having a part or a full copy. Once the resource parts have been found, the query node directly communicate with the hosting peers to obtain them.

P2P applications share most of distributed systems characterizing features, however they differ in a number of aspects, the most relevant of which are reported in the following. Each participant node in a P2P network acts as both a client and a server and is therefore able to share and search for resources. Also, due to the fact that no centralized server is required to achieve *all-to-all* communication and coordination, P2P systems are highly scalable and control is completely distributed. Heterogeneity is another feature of P2P applications, in that peers can be any kind of computing resource that can run the P2P software (*e.g.*, PCs, workstations, mobile devices, boards such as Raspberry PI and Intel Galileo). Finally, P2P systems are highly dynamic due to the constant joining and leaving of nodes.

1.2.1 Peer-to-peer benefits

P2P systems leverage computing resources with low costs and effectively disseminated information. For such reason, they meet the requirements of both individuals and businesses when dealing with applications concerning resource and data sharing. Resource sharing applications allow peers to access computing and storage capabilities available in the network. Data sharing applications allow users to access and modify data among each other. In this way, users become at the same time data producers and consumers. A request for a content is passed from peer to peer until it reaches a node having the required content. Such a peer sends the content to the requestor directly or through all peers that forwarded the request. Applications which benefit from P2P are digital content sharing and computationally demanding tasks. For example, scientific computation that tipically requires supercomputers can leverage the presence of a huge number of connected nodes sharing their resources to perform tasks. Other applications involve collaborative environments, storage, distributed databases and gaming.

1.2.2 Peer-to-peer challenges

P2P has a great potential in bringing benefits, but is still affected by a number of issues related to the aforementioned features. Free-riding (*i.e.*, peers that download files and do not allow uploading to minimize their own bandwidth utilization) is a major problem. Also, the dynamism for which nodes constantly join and leave the network hinders availability in that a resource may be available at some time, but not at others. Another aspect to be considered is that the same query, in different times, can be answered in different ways. Routing and resource discovery is another substantial aspect. For example Gnutella addresses the problem through message flooding which does not require metadata, but such solution is not efficient. On the other hand, completely relying on metadata poses the problem of defining such metadata to be sufficiently accurate. The interaction among peers also raises the trust problem, which can be addressed through reputation management techniques. However, such techniques require accountability and hinder privacy and anonymity.

1.3 Autonomic Computing

Autonomic Computing (AC) is a concept that brings together many fields of computing with the purpose of creating self-managing systems aiming at decreasing human involvement [17]. The term *autonomic* comes from biology in that, the autonomic nervous system of living beings takes care of unconscious reflexes (*i.e.,* bodily functions that do not require attention, such as size of the pupil adjustments, digestive functions, the rate and depth of respiration, and the dilatation or constriction of blood vessels). Without the autonomic nervous system, we would be constantly busy consciously adapting our body to its needs and the environment. The research community agrees upon using AC to describe systems in which decision making and resource management dynamically change to reflect the current environmental context [18].

In a self-managing autonomic system, the human operator does not directly control the system. Instead, he/she defines general policies and rules to guide the self-management process. IBM defined the following properties for such a process.

- *Self-configuring:* the system should dynamically adapt to changes occurring in the environment.
- *Self-healing:* the system should discover, diagnose and correct faults.

- *Self-optimizing:* the system should monitor, control and balance resource use to ensure the optimal functioning with respect to defined requirements
- *Self-protecting:* the system should proactively detect and protect from attacks.

Along with the aforementioned properties, autonomic systems must be aware of their resources (*self-awareness*) and of the environment they are acting in (*context-awareness*), should support multiple platforms (*openness*) and hide complexity to end-users [19].

Autonomic systems implement control loops with the purpose of monitoring resources, and autonomously try to keep parameters in a desired range. IBM has suggested a reference model for autonomic control loops named the MAPE- K (Monitor, Analyze, Plan, Execute, Knowledge) loop [20], used to describe the architectural aspects of autonomic systems. As presented by Fig. 1.3 [19], an autonomic system is composed of autonomic elements (AEs), which can be considered as software agents composed of two parts: a managed element (ME) and an autonomic manager (AM). MEs are system resources – software or hardware – monitored by sensors. AMs enact the MAPE-K loop as described in the following. The monitor function aggregates, correlates and filters sensed data to find a symptom needing analysis. The analyze function performs complex analysis, based on knowledge data on symptoms provided by the monitor function. If the system behavior should change, a request is passed to the plan function which defines a procedure to enact the desired changes in the way resources are managed. Such changes are intended to achieve the system's goals. The change plan is then passed to the execute function, which employs effectors to produce the resource management changes. The shared knowledge base used to perform data analysis is created by the monitor function and updated by the execute function.

1.3.1 Autonomic computing benefits

The autonomic computing approach provides several benefits such as improved quality of service and reduced ownership, deployment and maintenance costs. Moreover,

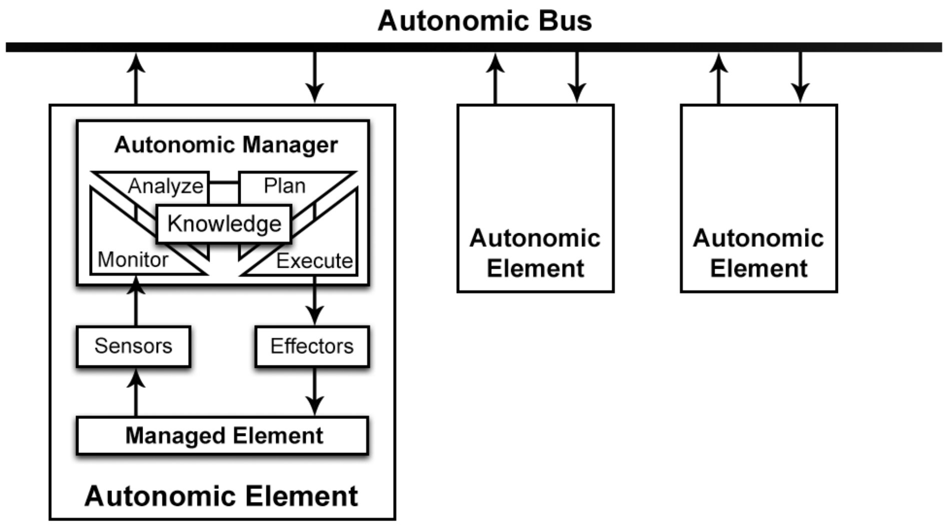


Figure 1.3: Autonomic Element architecture.

systems self-adapt to changes happening in the environment, support different platforms and are more stable through automation. As less personnel is needed to manage such systems, the chance of human errors is also reduced.

1.3.2 Autonomic computing challenges

Autonomic computing is characterized by a number of issues such as the implementation of self-management policies. Traditional systems are administered by IT experts who manually manage configuration, optimization, protection and healing. Transferring human knowledge to autonomic systems is a very challenging task. The definition of a robust learning procedure involves observing such experts and recording their activities' traces. Another challenge is the definition of the way AEs relate, interact and coordinate. Finally, since autonomic systems are deeply complex, ensuring robustness is challenging as each component may represent a single point of failure hindering the whole system functionality.

Chapter 2

Related Work

This work integrates MCC and AC principles to leverage the advantages of both paradigms. In fact, we believe autonomic principles should drive MCC systems where mobile devices monitor themselves and take offloading decisions. In this chapter we present the most interesting approaches for both computational paradigms.

2.1 Autonomic Computing approaches

Among available implementations of the MAPE-K loop (presented in the Introduction), the *Autonomic Computing Toolkit (ACT)* is a collection of self-managing autonomic technologies [21]. The toolkit is based on the dual concepts of *Managed Resources*, which represent network nodes or software components, and *Autonomic Managers*. Resources monitor the environment and are able to detect and report events to a manager. Also, resources take administrative actions in response to managers' requests. Managers oversee resource operations and implement administration policies and business logic. The ACT defines a data format called *Common Base Event (CBE)* used for event communication. At a conceptual level, ACT can be integrated with client/server architectures, in that any network device can be modeled as a managed resource, and server or gateway nodes can be configured as managers.

The Agent Building and Learning Environment (ABLE) [22] is a Java-based toolkit for developing and deploying hybrid intelligent agent applications. *AbleBeans* are standard JavaBeans components connected with each other. ABLE offers autonomic management in the form of a multi-agent architecture, in which the *autonomic manager* is an agent (*i.e.*, an AbleBean provided with sensors and effectors to interact with its environment) or a set of agents.

Kinesthetics eXtreme (KX) [23, 24] is another implementation of the MAPE-K loop, whose main purpose is the addition of autonomic properties to legacy systems. KX is the implementation of a framework the authors defined for collecting and interpreting application-specific behavioral data at runtime. Such a monitoring framework can be used with a feedback loop that automatically performs repairs and reconfigurations.

2.2 Mobile Cloud Computing approaches

Mobile augmentation through software approaches differ in the type of leveraged resources. Cloud resources are more frequently used with respect to distant servers or nearby surrogates due to issues in security, elasticity and resource availability. The approaches can be classified into six groups: remote execution, remote storage, multi-tier programming, live cloud-streaming, resource-aware computing and fidelity adaptation [2]. Approaches similar to the work of this Thesis belong to the *remote execution* category. In this chapter, we present the most interesting ones.

In remote execution approaches, demanding applications are entirely or partly executed on resource-rich remote devices. Slingshot [25] is an architecture for deploying application replicas on surrogate computers located near hotspots. Scavenger [26] is also a system allowing the development of highly distributed and parallel cyber-foraging applications.

While cyber-foraging implies a replica of the whole application, other approaches only migrate demanding components. In such cases, partitioning is of capital importance for the offloading process, as it involves the definition of which components can be partitioned, when partitioning should take place and which parts should be executed locally or remotely [27]. Partitioning can be addressed at design time, at runtime or in a hybrid way.

2.2.1 Design-time partitioning

In design-time partitioning approaches, programmers identify which parts of the application are computationally intensive and can be remotely offloaded. Such parts get marked in a way that depends on the programming language and, at runtime, the system dynamically decides whether to offload them or not based on policies. Defining such policies is a demanding and difficult task in that it requires a deep knowledge of the execution environment to adapt to diverse situations. For this reason, the design-time partitioning approach saves native resources, but is not optimal.

Spectra [28] is a remote execution system enabling mobile applications to leverage the processing power of static servers. The system monitors both application resource usage and the availability of resources in the environment, and dynamically determines where to execute application components. Decisions are taken balancing performance and energy conservation competing goals. The major drawback of Spectra is that application developers must explicitly modify their applications to use it, thus hindering software maintenance and portability.

Chroma [29] subsumes the functionalities of Spectra and addresses its shortcomings by separating application-level adaptive policies from decision-making and runtime policy enforcing.

2.2.2 Runtime partitioning

Runtime partitioning approaches rely on algorithms which monitor the application execution to identify compute-intensive parts and dynamically decide where to execute them.

Murarasu *et al.* [27] described the requirements and the design of a context-aware middleware enabling applications to automatically switch between local and remote services, and present a programming model including service life-cycle and state migration.

Abebe *et al.* [30] presented a distributed approach for adaptive offloading. In the proposed application representation, each device maintains a graph consisting only of components in its memory space, while maintaining abstraction elements for components in remote devices. This approach reduces the overhead from storing, updating and partitioning complete application graphs on each device, which limits their utility and scalability in resource constrained mobile environments.

March *et al.* [31] proposed μCloud, a framework which models a rich mobile application as a graph of components distributed onto mobile devices and a cloud infrastructure.

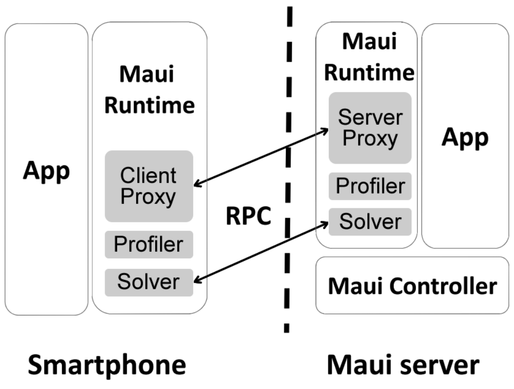


Figure 2.1: MAUI architecture.

MAUI is a system that enables fine-grained code offloading to a remote cloud infrastructure through runtime analysis [32]. In Fig. 2.1, MAUI's architecture is presented. Developers are enabled to produce an initial partitioning of applications by annotating as remoteable those methods and/or classes that the MAUI runtime should consider for offloading to a MAUI server. The authors suggest that all methods should be marked as remoteable except the ones implementing the GUI, the ones interacting with I/O devices and the ones interacting with external components that would be affected by re-execution. MAUI is currently designed only to support applications written for the Microsoft .NET Common Language Runtime (CLR). All CLR applications are compiled to the Common Intermediate Language (CIL) whose executable is dynamically compiled by the CLR at execution time, thus achieving platform independence. To identify remoteable methods in the CLI, MAUI uses the custom attributes feature of the CLR. Such attributes are metadata that annotate specific code elements, such as methods or classes, and are included in the compiled .NET CLR executables. The developer edits the application's source code by adding the *[Remoteable]* attribute to each method that can execute remotely. The MAUI runtime uses the .NET Reflection API to automatically identify which methods have been marked. At compile time, MAUI generates a wrapper for each marked method. Such a wrapper follows the original method's type signature, and adds one additional input parameter used to transfer the state from the smartphone to the MAUI server, and one additional return value used to transfer the application state back to the smartphone. The approach adopted by the authors for state transfer is to serialize all variables of the current object, the state of static classes and any public static variable using the .NET built-in support for XML-based serialization. To optimize the overhead of state transfer, just deltas are shipped rather than the entire state. At compile time, MAUI generates two proxies, one running on the smartphone and one on the MAUI server. The role of such proxies is to implement decisions made by the MAUI solver which decides whether each method should be executed locally or remotely, based on input from the MAUI profiler. For calls that transfer control to a remote server, the local proxy performs state serialization before the call and deserialization of the returned state. When a remoteable method currently executing on the MAUI server invokes a method which is not remoteable, the server-side proxy performs the necessary serialization and transfers control back to the smartphone. At runtime, before each method is invoked MAUI determines whether the method should run locally or remotely based on three factors: the device's energy consumption characteristics, the program characteristics (*e.g.*, the running time and resources needed by individual methods) and network characteristics (*e.g.*, the bandwidth, latency, and packet loss). The MAUI profiler is the component which performs device profiling by instrument-ing each method to measure its state transfer requirements. The profiler measures the device characteristics at initialization time, and continuously monitors the program and network characteristics in that these can often change and force MAUI to make wrong decisions. However, such a task is challenging in that applications are not deterministic. Each subsequent invocation of a method can take a different code path, leading to a different running duration and energy profile. Thus, MAUI uses past invocations of a method as a predictor of future invocations. The MAUI solver uses data collected by the profiler as input to an optimization problem that determines where each remoteable method should execute. Decisions must be globally optimal rather than locally optimal (*i.e.*, relative to a single method invocation). Consider for example a face recognition application composed of a certain number of methods. The remote execution of each method is more expensive than local execution, however remote execution can save energy if all methods are remoted.

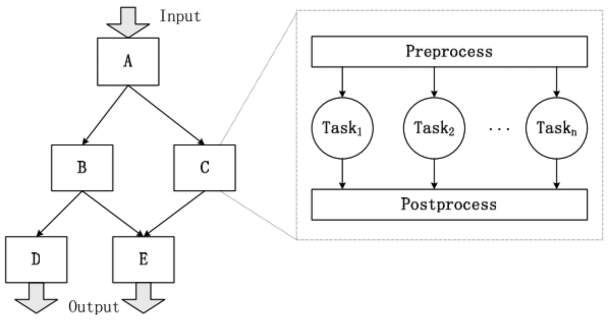


Figure 2.2: Serendipity job model.

Serendipity [33] enables a mobile user to leverage remote computational resources available on other mobile systems to enhance performance and conserve energy. The approach relies on the collaboration among mobile devices for task allocation and task progress monitoring functions. The authors designed a job model to simplify the data flow among tasks for which each job is composed of *PNP-blocks*. As illustrated by Fig. 2.2, a block is composed of a *pre-processing* program which processes the input data and passes them to the tasks, *n* parallel task programs and a *post-processing* program which processes the output of all tasks. All *pre-process*

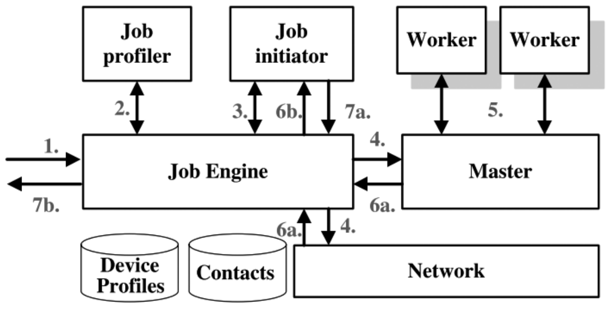


Figure 2.3: Serendipity architecture.

and *post-process* programs are executed on one initiator device, while parallel tasks are executed independently on other devices. As presented by Fig. 2.3, a Serendipity node includes a *job engine* process, a *master* process and several *worker* processes, whose number can be, for example, the number of cores or processors of the node. Each node constructs its device profile and shares and maintains the profiles of encountered nodes. Such profiles include the execution speed, estimated by running synthetic benchmarks, and the energy consumption model. When device and execution profiles are combined, it is possible to estimate the jobs' execution time and energy consumption on every node. Such an information is then used for task allocation. All jobs are represented by a directed acyclic graph (DAG) of *PNP-blocks*. To submit a job, a user needs to provide a script specifying the job DAG, the programs and their execution profiles for all *PNP-blocks*, and the input data to the *job engine*. The script is submitted to the *job profiler* which performs basic checking and constructs a complete job profile consisting of tasks' execution time and energy consumption on every node. The generated job profile is then used to decide the allocation of its tasks among mobile devices. The job engine then starts a new *job initiator* responsible for the new job. *PNP-blocks* start by running their pre-process programs on a local worker and assigning a time-to-live (TTL), a priority and a worker to every task. The TTL specifies the time before which its results should be returned. If a task misses its TTL, it should be discarded, while a copy executes locally on the initiator's mobile device. Finally, the tasks are sent to the *job engine* which is primarily responsible for disseminating and scheduling task execution for the local *master*. When two mobile nodes encounter, they first exchange metadata including their device profiles, their residual energy and a summary of carried tasks. Based on such an information, the job engine decides whether to disseminate a task to the encountered node or to execute it locally. Upon receiving a task from the job engine, the *master* starts a worker for it and, when the task finishes, sends the output back to the job initiator.

The main difficulties when using Serendipity reside in the fact that developers have to generate DAGs representing jobs and profile devices.

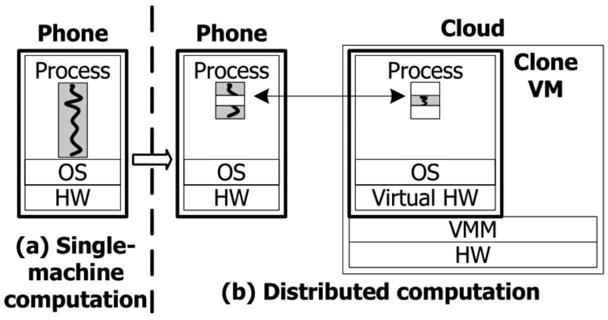


Figure 2.4: CloneCloud system model.

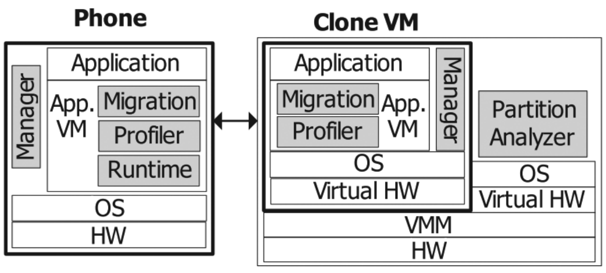


Figure 2.5: CloneCloud architecture.

CloneCloud [34] is a framework which aims at enabling unmodified mobile applications to offload parts of execution from mobile devices to device clones on a computational cloud. The system uses static analysis and dynamic profiling to automatically partition applications at a fine granularity. The optimization of execution time and energy use is based on the computation and communication environment. At runtime, the application partitioning happens through migrating, at a chosen point, a thread from the mobile device to the clone in the cloud, executing there for the remainder of the partition, and re-integrating the thread back to the mobile device. The concept of CloneCloud is that a single-machine execution is transformed into a distributed execution optimized for the network connection towards the cloud, the processing capabilities of the device and cloud, and the application's computing patterns, as presented by Fig. 2.4. The design goal of CloneCloud is to allow flexible fine-grained partitioning while hiding the partitioning complexity to developers. In Fig. 2.5 the CloneCloud architecture is presented. The authors implemented a prototype that meets the aforementioned goal by rewriting an unmodified application executable, so that individual threads migrate from the mobile device to a clone in the cloud at automatically chosen points. Functionalities that remain on the mobile device keep executing, but block if they attempt to access migrated state. Migrated threads eventually return back to the mobile device, along with remotely created state that merges back into the original process. The choice of where to migrate is made by a partitioning component, which uses static analysis to discover constraints on possible migration points, and dynamic profiling to build a cost model for execution and migration. A mathematical optimizer chooses migration points that optimize the objective (*e.g.*, total execution time or mobile-device energy consumption), given the application and the cost model. The output of the optimizer is then used to build a database of pre-computed partitions of the binary, whose purpose is to determine which parts should be executed remotely and which locally.

ThinkAir [35] is a framework that tackles MAUI's and CloneCloud's limits. It addresses MAUI's lack of scalability by creating VMs of a complete smartphone system on the cloud, and uses an online method-level offloading to remove restrictions on applications, inputs and environmental conditions that CloneCloud induces. Fig. 2.6 presents the framework which consists of three major components: the *execution environment*, the *application server* and the *profilers*. ThinkAir provides a library that, coupled with the compiler support, allows annotating any method to be considered for offloading with the @*Remote* tag. The ThinkAir *code generator* takes

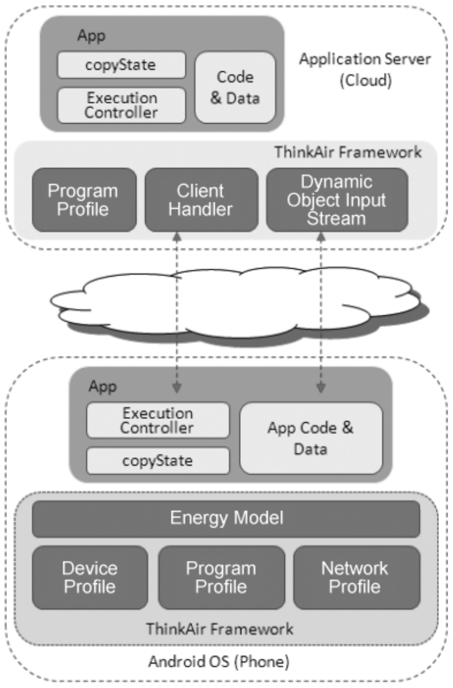


Figure 2.6: ThinkAir framework overview.

the source file and generates remoteable method wrappers and utility functions, making it ready for use with the framework. Method invocation is done via the *execution controller*, which detects if a given method is a candidate for offloading and handles all the associated profiling, decision making and communication with the *application server* without the developer needing to be aware of the details. The *compiler* is a key part of the framework and includes two components: the *Remoteable Code Generator* and the *Customized Native Development Kit (NDK)*. The former translates the annotated code as previously described. The latter provides native code support addressing the fact that most current mobile platforms are ARM-based, while cloud infrastructures use x86 nodes. The *execution controller* drives the execution by deciding whether to offload or continue local execution of each remoteable method. Such a decision depends on data regarding the current environment as well as data learnt from past executions. When a method is encountered for the first time, the decision is based only on environmental parameters and, at the same time, profilers start collecting data. Therefore, if and when the method is subsequently encountered, the decision on where to execute it is based on past invocations. The *application server* manages the cloud side of offloaded code and consists of three main parts: a *client handler*, a *cloud infrastructure*, and an *automatic parallelization component*. The *client handler* executes the ThinkAir communication protocol to manage client connections, receive and execute offloaded code, and return results. To manage client connections, the *client handler* registers when a new application connects and, if the latter is unknown to the application server, the *client handler* retrieves such an application from the client and loads required classes and libraries. Then, the server waits for execution requests from the client. If there is no request for further computational power, the *client handler* proceeds as the client would, that is, the remoteable method is called using Java reflection and the result is sent back. If instead the client asks for extra computational power, the *client handler* resumes a more powerful clone and delegates the task to it. If the client asks for parallel execution on more clones, the *client handler* resumes the necessary clones, distributes the task among them, and collects and sends results back to the client.

2.2.3 Hybrid partitioning

Due to the heterogeneity of execution environments, design-time partitions are not guaranteed to be the best solution for all possible scenarios. On the other hand, run-time partitioning allows a system to adapt according to the characteristics of the current execution environment, but profiling and monitoring introduce a remarkable overhead. Huerta-Canepa and Lee [36] proposed an hybrid partitioning approach to perform offloading based on the execution history of applications. When an application has to be executed on a resource-constrained device, the system checks whether previous executions information is available. If such is enough to perform statistical sampling, and based on current device's conditions, the approach chooses one of four possible actions. *No action* indicates that an application will run normally. *Profile*

Only means that an application gets profiled to update historical data, but no further action is taken. *Static Offloading* triggers offloading before the execution of the main class of an application. *Dynamic Offloading* profiles an application and triggers offloading only if necessary. The process is performed for each resource used by the application.

Chapter 3

Networked Autonomic Machine

As discussed in the Introduction, while the ever increasing communication capabilities of mobile devices make viable offloading computation and storage to remote devices and infrastructures, several issues and challenges hinder the full realization of MCC systems. Among those, significant are the lack of an agreed upon conceptual model for MCC systems; the fact that most of current applications are statically partitioned; the possibility of rapid changes in network conditions and local resource availability; privacy and security concerns related to storing user data on a remote cloud [4, 37, 10]. Also, as the task to be performed and the execution context influence the offloading approach [10], autonomic computing appears as a promising technique to increase the robustness and flexibility of MCC systems. In particular, autonomic policies based on continuous resource and connectivity monitoring may help automate context-aware decisions for computation offloading.

The need for modeling large-scale distributed systems provided with autonomic mechanisms led to the definition of the *Networked Autonomic Machine (NAM)* framework [38, 39], which is also suitable for MCC, as it supports code, execution state and data mobility concepts [40, 41].

3.1 NAM-based model

A system of NAMs is a loosely connected network of hardware/software entities, which provide or consume services. In a NAM network, each device can host one or more NAMs. A NAM represents a container of computational entities and data. Computational entities are service threads exploiting functionalities provided by libraries called *functional modules*. Both *application data* and *awareness data* are considered. The former is used by applications running on the NAM node, while the latter provides information about the environment in which the NAM is running (*e.g.*, sensor information and context events) or about the status of the NAM itself.

More formally, a NAM is represented as a tuple $nam = \langle nid, \mathscr{R}, \mathscr{F}, \mathscr{P} \rangle$, where *nid* is the NAM identifier, $\mathscr{R}$ is a set of physical *resources*, $\mathscr{F} = \{f_1, \ldots, f_m\}$ is a set of functional modules, and $\mathscr{P}$ is a set of self-management policies. Resources are, for example, CPU cycles, storage space, network interfaces. Each NAM is allowed to directly access its own resources. Instead, remote resources (*i.e.*, resources managed by another NAM) are not directly accessible, as a NAM can only interact with services exposed by other NAMs. Data are not considered as a resource, and it is assumed it is always stored within functional modules and moved accordingly. The state of a NAM consists of sets $\mathscr{R}$, representing available resources, and $\mathscr{F}$, describing functional modules that currently reside on it. Autonomic policies are a crucial means to support MCC, as they alleviate mobile users from manually starting/stopping applications, or application modules, when their execution becomes too demanding in terms of local resources. Specifically, a policy is expressed as an *Event-Condition-Action (ECA)* rule in the form (ev, co, act). The occurrence of *ev* event triggers the evaluation of the corresponding *co* condition and, in case of positive evaluation, the *act* action is performed.

3.1.1 NAM computational entities

NAM computational entities are *functional modules* and *services*. A functional module is a specialized module represented as a tuple $f = \langle fid, \mathscr{S}, \mathscr{P}_f, \mathscr{D}, \mathscr{T} \rangle$, where *fid* is the functional module identifier, $\mathscr{S}$ is a set of bindings from service names to methods of f implementing them, $\mathscr{P}_f$ is a tuple containing functional policies of the module (*i.e.*, policies that define module-specific actions to be taken when particular conditions hold), $\mathscr{D}$ is a set of data available to the module, and $\mathscr{T}$ is a set of threads the module is running.

A *service* is an entry point for a functional module, which has the role of aggregating functions and data to provide computational tasks. More specifically, functions hosted by functional modules are accessed by other (local or remote) functional modules via services. To this purpose, when a functional module receives a service request, it identifies the corresponding local or remote method through bindings in $\mathscr{S}$, and subsequently creates a thread to execute it.

Events represent another form of entry point, but they differ from services in that, while a service request triggers a thread execution, an event triggers a policy evaluation and possibly, a functional or self-management action. In fact, while services are specifically devised to support client-server communication, events also enable publish-subscribe interactions. Functional module policies $\mathscr{P}_f = \langle \mathscr{P}_o, \mathscr{P}_l, \mathscr{P}_r \rangle$ are structured in three parts: $\mathscr{P}_o$ are the *on-site* policies, active when the module is executing locally on the device, while $\mathscr{P}_l, \mathscr{P}_r$ are policies activated on the *local* and *remote* NAMs respectively, when the module is offloaded. The existence of local and remote policies during offloading is motivated by the need of evaluating events both locally and remotely. An example of a local event is the detection of when the connection quality gets lower than a specified threshold. Such an event may trigger the reintegration of an offloaded module. Similarly, a remote event can generate when resources become scarce, thus triggering the decision of sending the module back to the owner.

3.2 Mobility actions

Mobility is a central aspect of NAM networks, as it allows the dynamic reconfiguration of the system by moving functional modules and services from NAM to NAM.

The framework supports five mobility actions: offload, back, go, migrate, and copy. Fig. 3.1 [41] presents four scenarios in which such actions can be used.

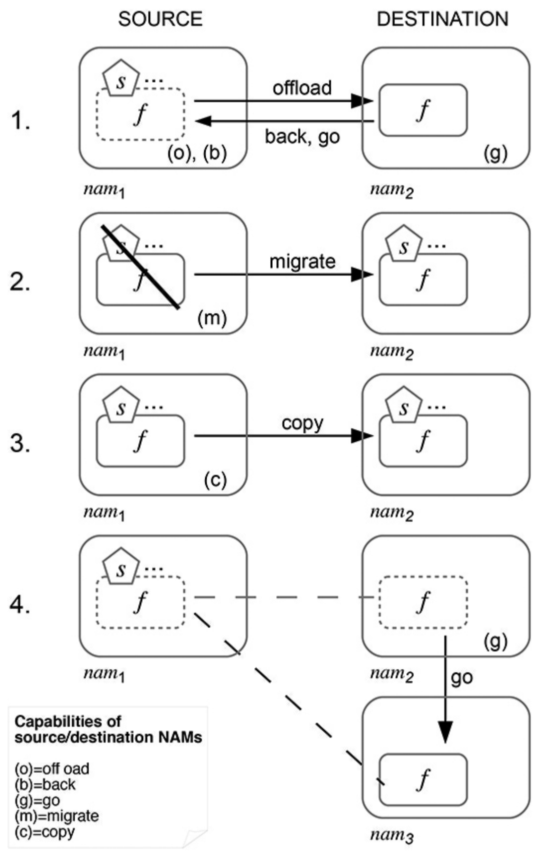


Figure 3.1: Mobility actions.

In the first scenario, nam_1's resources (such as battery or cpu cycles) become scarce and therefore, according to its internal policies, nam_1 decides to move the code of functional module f to nam_2 through an offload action. Thus, the resource-consuming elements of f (*i.e.*, data $\mathscr{D}$ and running threads $\mathscr{T}$) move to nam_2. Such elements get regulated by specific $\mathscr{P}_r$ policies, while $\mathscr{P}_l$ policies are activated and enforced locally on nam_1. Therefore, f stops consuming resources of the source and starts consuming the ones of the destination. The entry points of f (*i.e.*, the services specified in $\mathscr{S}$) remain on nam_1. Such a choice is motivated by the need for full offloading transparency, with respect to local and remote modules that use services of f. The operation requires service bindings in $\mathscr{S}$ on nam_1 to be modified to redirect service requests to nam_2. If necessary, nam_1 can request to terminate the f offloading by executing a back action, which reintegrates f on nam_1 and consistently updates $\mathscr{S}$ and active policies. Finally, in case nam_2 decides it cannot provide further hosting for f (*e.g.*, nam_2 is a cloud service and nam_1 is running out of credit), it can execute a go action which reintegrates f on nam_1.

The second scenario considers an autonomic functional module f, such as a crawler. In this case, the whole functional module (including services and service bindings) can request to be moved to another NAM. Nam_1 moves f to nam_2 by executing a migrate action, after which no part of f is available on nam_1. Clearly, such an action requires to update the $\mathscr{F}_1$ set of functional modules on nam_1, as well as the $\mathscr{F}_2$ set of functional modules on nam_2.

The third scenario considers operations such as downloading applications or libraries. Upon receiving a request from nam_2 for module f, nam_1 copies it on nam_2 through a copy action. As a consequence, nam_2 can access the services of f locally, without relying on nam_1. The action modifies the $\mathscr{F}_2$ set of functional modules on nam_2.

Finally, the fourth scenario considers operations that move offloaded modules, for example to perform load-balancing. In Fig. 3.1, nam_2 hosts a module offloaded by nam_1 and decides it cannot offer further hosting. Thus, it moves f to nam_3 through a go action which moves all elements of f from nam_2 to nam_3 and updates $\mathscr{S}$ on nam_1 (dashed lines in the figure represent such an update).

Note that for back, go, migrate and offload actions, the execution of f module's $\mathscr{T}$ threads is suspended and recovered on the remote location. Similarly, local data $\mathscr{D}$ of the module is moved to the remote location. On the contrary, in a copy action f has no track of previous execution on nam_1. Therefore, $\mathscr{D}$ and $\mathscr{T}$ are set as empty on nam_2. Also, a NAM can perform copy, go, migrate and offload mobility actions on a local functional module only, and back on a remote one only.

3.3 NAM formalization

As part of the PhD research activities, the NAM framework has been provided with a formalization in terms of a *transformational operational semantics* in order to fill the gap between its implementation – NAM4J – and its conceptual definition. We used the *Kernel Language for Agents Interaction and Mobility* (KLAIM) [42], which is a linguistic formalism specifically designed to model distributed systems consisting of several mobile components. The interaction among components happen through multiple distributed shared memories, called *tuple spaces*. KLAIM primitives allow programs to distribute/retrieve data and processes to/from the nodes of a network, thus enabling for data and code mobility. For further information on KLAIM check appendix A. The formalization process clarified and allowed us to refine the NAM framework with specific focus on MCC features.

Most of AC proposals in the literature concern full-fledged programming languages rather than foundational models. Some proposed formalisms, as *e.g.* in [43, 44], are inspired by chemical and biological phenomena. A formalism based on KLAIM, close to programming languages and following a *process calculi* approach, is SCEL [45]. Although SCEL is equipped with constructs for dealing with autonomicity, it mainly provides communication primitives to manage ensembles. The latter are not relevant for our study and complicate the operational semantics. In fact, SCEL is not currently equipped with verification tools, which we want to use to analyze MCC-based applications. Therefore, we selected KLAIM as a basis for the formalization process as it allows for the convenient modeling of autonomic features and supports strong and weak mobility mechanisms [46]. A combination of both mobility and autonomicity is necessary to properly model MCC scenarios. Moreover, KLAIM comes with software tools that support various forms of analysis. Intuitively, the NAM formalization associates a KLAIM term to every NAM construct.

This section discusses how, from an operational point of view, a NAM network can be defined in terms of a KLAIM network. In particular, the aim of providing the semantics of the NAM framework in terms of the KLAIM formal language is to clarify relationships among functional modules, related services and the underlying middleware. For the sake of readability, in this section we omit the target **self** from KLAIM actions by writing, *e.g.*, $\mathbf{in}(T)$ in place of $\mathbf{in}(T)@\mathbf{self}$.

A NAM network consisting of a set of nodes $\{nam_1, \ldots, nam_m\}$ can be rendered in KLAIM as a net

$$nid_1 ::_{\rho_1} (C_{TS}^1 \mid C_P^1) \;\|\; \ldots \;\|\; nid_m ::_{\rho_m} (C_{TS}^m \mid C_P^m)$$

where nid_i is the identifier of nam_i and ρ_i stands for $\{\mathbf{self} \mapsto nid_i\}$. Intuitively, each NAM $\langle nid, \mathscr{R}, \mathscr{F}, \mathscr{P} \rangle$ is modeled as a KLAIM node having tuple space C_{TS} and running processes C_P.

Tuples stored in C_{TS} represent data local to functional modules in $\mathscr{F}$, resources available in $\mathscr{R}$, messages denoting service requests or events, code of functional modules in $\mathscr{F}$, and commands to instrument mobility actions supported by the framework.

The first field of each tuple is a tag string indicating the tuple's role (*e.g.*, tuple $\langle \mathsf{srvReq}, sid, data, nid_{SRC} \rangle$ denotes a service request containing the identifier of the requested service, input data and the identifier of the NAM invoking the service).

Processes in C_P perform computational tasks and enact NAM self-management policies. Such processes are defined as a parallel composition

$$Disp \;\mid\; PMH \;\mid\; F_1 \;\mid\; \ldots \;\mid\; F_k\ ,$$

where:

- *Disp* is a *dispatcher* of service requests to appropriate functional modules;
- *PMH* is the *policy and mobility handler* in charge of enforcing NAM policies in $\mathscr{P}$ and executing mobility commands;
- F_j includes processes which model the functional module f_j in $\mathscr{F}$ (*i.e.*, the service handler (*SH*) and the policy handler (*PH*) of the functional module), whose identifier is *fid* and has a set (T) of threads, each managing a specific service request
$$SH_{fid} \;\mid\; PH_{fid} \;\mid\; T_{fid}^1 \;\mid\; \ldots \;\mid\; T_{fid}^h$$

In the following, details concerning the aforementioned processes are provided.

3.3.1 Control tuples

NAM's synchronization and mobility actions are associated to a set of control tuples, described in the following. Services are bound to functional modules through identifiers and may be located on a local or a remote NAM. A service is implemented within a functional module by process *Proc*.

$\langle \mathsf{srvBinder}, sid, fid, nid \rangle$ (a service binder including the functional module's id and location)
$\langle \mathsf{srvImplem}, sid, fid, Proc \rangle$ (a service implementation in a specific functional module)

Accessing a service happens through a *service request* which gets dispatched to a specific functional module *fid* through a *service assignment*, if the module is local, or through a *remote service assignment*, if the module is remote.

$\langle \mathsf{srvReq}, sid, data, nid \rangle$ (a service request including service name, data, and source NAM)
$\langle \mathsf{srvAssign}, sid, fid, data, nid \rangle$ (a service assignment to a local functional module)
$\langle \mathsf{remoteSrvAssign}, sid, fid, data, nid \rangle$ (a service assignment to a remote functional module).

When a NAM receives an offloaded functional module, it is aware of the sender's identity *nid*, and is therefore able to send the module back to the owner, if needed:

$\langle \mathsf{offloaderNAM}, fid, nid \rangle$ (*nid* of *fid*'s offloader)

Mobility actions are initiated by five possible *mobility requests*, issued by NAM or functional module policies:

$\langle \mathsf{backReq}, fid, nid \rangle$ $\langle \mathsf{copyReq}, fid, nid \rangle$ $\langle \mathsf{goReq}, fid, nid \rangle$
$\langle \mathsf{migrateReq}, fid, nid \rangle$ $\langle \mathsf{offloadReq}, fid, nid \rangle$

When the mobility handler reacts to mobility requests, it sends appropriate mobility commands to the service handler of the corresponding functional module,

$\langle \mathsf{backSH}, fid, nid \rangle$ $\langle \mathsf{copySH}, fid, nid \rangle$ $\langle \mathsf{goSH}, fid, nid \rangle$
$\langle \mathsf{migrateSH}, fid, nid \rangle$ $\langle \mathsf{offloadSH}, fid, nid \rangle$ $\langle \mathsf{remoteBackSH}, fid \rangle$

and to its policy handler,

$\langle \mathsf{backPH}, \mathit{fid}, \mathit{nid}\rangle$ $\langle \mathsf{copyPH}, \mathit{fid}, \mathit{nid}\rangle$ $\langle \mathsf{goPH}, \mathit{fid}, \mathit{nid}\rangle$
$\langle \mathsf{migratePH}, \mathit{fid}, \mathit{nid}\rangle$ $\langle \mathsf{offloadPH}, \mathit{fid}, \mathit{nid}\rangle$ $\langle \mathsf{remoteBackPH}, \mathit{fid}, \mathit{nid}\rangle$

Running threads $\langle \mathsf{thread}, \mathit{fid}, \mathit{tid}\rangle$ are associated to a functional module and have their own unique identifier *tid*, used when migrating or offloading the module. When a migrate or offload action is performed, mobility requests $\langle \mathsf{moveThread}, \mathit{tid}\rangle$ are issued for each thread. Thread code must be correctly instrumented so that it can handle mobility requests in a proper manner.

3.3.2 NAM control

The dispatcher, whose main purpose is the selection of an appropriate functional module upon receiving a service request, is defined as a process

```
$Disp\ =$
  $\mathbf{in}(\mathsf{srvReq}, ?sid, ?data, ?nid_{SRC});$
  $\mathbf{read}(\mathsf{srvBinder}, sid, ?fid, ?nid_{IMP});$
  $\mathbf{if}\ (nid_{IMP} == \mathbf{self})$
    $\mathbf{then}\{\mathbf{out}(\mathsf{srvAssign}, sid, fid, data, nid_{SRC})\}$
    $\mathbf{else}\ \{\mathbf{out}(\mathsf{remoteSrvAssign}, sid, fid, data, nid_{SRC})@nid_{IMP}\};$
  $Disp$
```

Such a process cyclically reads and consumes a service request, finds the NAM which hosts the functional module implementing the service (which can either be the NAM hosting the dispatcher itself or a remote NAM), and sends a service assignment to such a NAM. More specifically, *service binder* tuples $\langle \mathsf{srvBinder}, sid, fid, nid_{IMP}\rangle$, stored on NAMs, are used to identify through *pattern-matching* the NAM nid_{IMP} which provides the implementation of module *fid* exposing service *sid*. Depending on whether nid_{IMP} is local or remote, either a *local service assignment* (tagged by srvAssign) or a *remote service assignment* (tagged by remoteSrvAssign) is generated.

Mobility commands are managed by the *mobility handler* MH

$$MH = CH + MiH + OH + BH + GH$$

where *CH* is the copy action handler, *MiH* is the migrate action handler, *OH* is the offload action handler, *BH* is the back action handler, and *GH* is the go action handler.

The following steps allow moving functional module's threads during offloading or migration. The identifier of each thread associated to the module is retrieved and deleted, a moveThread message is sent (thus relying on the thread ability to react to such requests), and threads are registered on the remote location.

Policies are managed by the *policy handler*, which is rendered as a choice composition of processes modeling event-condition-action rules of $\mathscr{P}$ NAM policies. In particular, the *ev* event (retrieved by **in**) triggers the execution of P_{act} process, which enacts the *act* action if *co* condition is satisfied.

The process that models the policy and mobility handler of a NAM is

$$PMH = MH + \sum_{(ev,co,act)\in P_n} \mathbf{in}(\mathsf{event}, ev);\ \mathbf{if}\ (co)\ \mathbf{then}\ \{P_{act}\};\ PMH$$

The *PMH* component enacts mobility actions or policies in a mutually exclusive way. Thus, policy and mobility handlers only process one event at a time to avoid interferences among the executions of different mobility actions and policy evaluation.

3.3.3 Functional module control

Every functional module f is provided with a service handler SH_{fid} which reacts to service assignments by creating a thread that serves the associated request and changes state accordingly. The following KLAIM code models such a behavior.

$SH_{fid} =$

$\quad \mathbf{in}(\mathsf{srvAssign}, ?sid, fid, ?data, ?nid_{SRC});$

$\quad\quad START\_THREAD(sid, fid, data, nid_{SRC});$

$\quad\quad SH_{fid}$

$\quad + \mathbf{in}(\mathsf{copySH}, fid, ?nid_{DST}); \mathbf{eval}(SH_{fid})@nid_{DST};$

$\quad\quad SH_{fid}$

$\quad + \mathbf{in}(\mathsf{migrateSH}, fid, ?nid_{DST}); \mathbf{eval}(SH_{fid})@nid_{DST}$

$\quad + \mathbf{in}(\mathsf{offloadSH}, fid, ?nid_{DST}); \mathbf{eval}(RSH_{fid})@nid_{DST};$

$\quad\quad LSH_{fid}$

Upon receiving a service assignment (srvAssign) for *fid*, the service handler creates a thread including the *sid* service identifier, the *fid* module identifier, *data* for the computation and the nid_{SRC} client identifier. In case of a copy request (copySH) for *fid* to nid_{DST} destination, the service handler copies itself to nid_{DST} by using the **eval** action, and returns to its previous state. In case of a migrate request (migrateSH), the service handler behaves similarly, however it stops its execution. An offload request (offloadSH) is instead managed in a different way, in that it starts a *remote* service handler RSH_{fid} located on nid_{DST} and then switches to execute a *local* service handler LSH_{fid}. The KLAIM code for the aforementioned processes is reported in the following.

$LSH_{fid} =$

$\quad \mathbf{in}(\mathsf{backSH}, fid, ?nid_{DST});$

$\quad \mathbf{out}(\mathsf{remoteBackSH}, fid)@nid_{DST};$

$\quad SH_{fid}$

$RSH_{fid} =$

$\quad \mathbf{in}(\mathsf{remoteSrvAssign}, ?sid, fid, ?data, ?nid_{SRC});$

$\quad\quad START\_THREAD(sid, fid, data, nid_{SRC});$

$\quad\quad RSH_{fid}$

$\quad + \mathbf{in}(\mathsf{remoteBackSH}, fid)$

$\quad + \mathbf{in}(\mathsf{goSH}, fid, ?nid_{DST}); \mathbf{eval}(RSH_{fid})@nid_{DST}$

After offloading, *Disp* dispatcher process forwards service requests to the remote NAM nid_{DST}. LSH_{fid} reacts to a back request (backSH) by notifying the remote NAM through a remoteBackSH request, and returns to (normal) SH_{fid} state. The remote service handler RSH_{fid} has three possible behaviors. The first one reacts to a remote service assignment (remoteSrvAssign) by creating a thread to serve the request and returns to its initial state. The second behavior receives a back request (remoteBackSH) and terminates the RSH_{fid} process. The last behavior reacts to a go request (goSH) to nid_{DST} by creating a remote service handler on nid_{DST} and terminates. The last behavior is activated only if the destination NAM nid_{DST} of the go action is not the functional module owner (*i.e.*, the offloader), otherwise RSH_{fid} would become SH_{fid}. Such a check is performed by the process triggering the goSH action, which is the mobility handler *MH*. After a go action, service requests are forwarded to the NAM where the remote service handler is active. Such a redirection requires the update of service bindings, which is managed by the mobility handler process.

Prior to presenting further details of mobility actions, code illustrating the way threads are created is reported.

```
$START\_THREAD(sid, fid, data, nid_{SRC}) =$
    $\textbf{read}(\text{srvImpl}, sid, fid, ?Code);$
    $tid := getFreshId();$
    $\textbf{out}(\text{thread}, fid, tid);$
    $\textbf{eval}(Code(tid, data, nid_{SRC}, fid))$
```

The implementation (*Code*) of a service in *fid* functional module is retrieved from a tuple tagged as srvImpl by using *sid* service identifier. Subsequently, a new thread identifier *tid* is created and registered as a *fid* thread – $Code(tid, data, nid_{SRC}, fid)$ – which is executed locally. The unique thread id registration phase is required to be able to retrieve and move running threads of a functional module when such a module is offloaded or migrated. The thread is expected to know the identifier of the service client (nid_{SRC}), to be able to reply to it, and its own identifier *tid*, to be able to unregister upon task completion and react to migrate/offload requests.

PH_{fid} policy handler manages policies by using triples (ev, co, act) in the on-site policy $\mathscr{P}_o$ of *fid*. Furthermore, it reacts to mobility actions identified by tuples whose tag is in $\{\mathsf{backPH}, \mathsf{copyPH}, \mathsf{goPH}, \mathsf{migratePH}, \mathsf{offloadPH}\}$ set. In particular, in the case of an offload request, the handler starts a remote policy handler RPH_{fid}, which executes the functional module remote policy $\mathscr{P}_r$, and switches to execute a local policy handler LPH_{fid} (which executes the functional module local policy $\mathscr{P}_l$). As for the service assignment handler, the policy handler presents a *normal* operation mode which executes policies in P_N on local events and handles mobility actions in a similar fashion.

$$
\begin{aligned}
&PH_{fid} = \\
&\quad \sum_{(ev,co,act)\in P_N} \mathbf{in}(\mathsf{event}, ev);\ \mathbf{if}\ (co)\ \mathbf{then}\ \{P_{act}\};\ PH_{fid} \\
&\qquad + \mathbf{in}(\mathsf{copyPH}, fid, ?nid_{DST}); \mathbf{eval}(PH_{fid})@nid_{DST}; \\
&\qquad\quad PH_{fid} \\
&\qquad + \mathbf{in}(\mathsf{migratePH}, fid, ?nid_{DST}); \mathbf{eval}(PH_{fid})@nid_{DST}; \\
&\qquad + \mathbf{in}(\mathsf{offloadPH}, fid, ?nid_{DST}); \mathbf{eval}(RPH_{fid})@nid_{DST}; \\
&\qquad\quad LPH_{fid}
\end{aligned}
$$

In *offloaded* mode instead, the policy handler is split into a local and a remote handler to react to both local and remote events.

$$
\begin{aligned}
&LPH_{fid} = \\
&\quad \sum_{(ev,co,act)\in P_L} \mathbf{in}(\mathsf{event}, ev);\ \mathbf{if}\ (co)\ \mathbf{then}\ \{P_{act}\};\ PH_{fid} \\
&\qquad + \mathbf{in}(\mathsf{backPH}, fid, ?nid_{DST}); \\
&\qquad\quad \mathbf{out}(\mathsf{remoteBackPH}, fid, nid_{DST})@nid_{DST}; \\
&\qquad\quad PH_{fid} \\
&RPH_{fid} = \\
&\quad \sum_{(ev,co,act)\in P_R} \mathbf{in}(\mathsf{event}, ev);\ \mathbf{if}\ (co)\ \mathbf{then}\ \{P_{act}\};\ PH_{fid} \\
&\qquad + \mathbf{in}(\mathsf{remoteBackPH}, fid, \_) \\
&\qquad + \mathbf{in}(\mathsf{goPH}, fid, ?nid_{DST}); \mathbf{eval}(RPH_{fid})@nid_{DST}
\end{aligned}
$$

3.3.4 Macros

Prior to describing the KLAIM code for all actions, macros which facilitate code reading and handle mobility operations are introduced. Such macros use the **while** construct with non-blocking variants of **in**/**read** as argument, to ensure that each tuple of interest is considered at least once. In case the argument is **read**, the **while** semantics ensures that each tuple is considered at most once. Notably, the **while** loops in macros certainly terminates, due to a disciplined use of tuples and appropriate boolean conditions on a number of their fields.

Service binders can be moved, copied and updated.

```
$MOVE\_BINDER(fid, nid_{DST}) =$
   $\textbf{while}\ (\textbf{inp}(\mathsf{srvBinder}, ?sid, fid, ?nid_{IMP}))$
        $\{\textbf{out}(\mathsf{srvBinder}, sid, fid, nid_{DST})@nid_{DST}\}$
$COPY\_BINDER(fid, nid_{DST}) =$
   $\textbf{while}\ (\textbf{readp}(\mathsf{srvBinder}, ?sid, fid, \textbf{self}))$
        $\{\textbf{out}(\mathsf{srvBinder}, sid, fid, nid_{DST})@nid_{DST}\};$
$UPDATE\_BINDER(fid, nid_{DST}) =$
   $\textbf{while}\ (\ \textbf{inp}(\mathsf{srvBinder}, ?sid, fid, ?nid_{IMP})\ \&\&\ nid_{IMP} != nid_{DST}\ )$
        $\{\textbf{out}(\mathsf{srvBinder}, sid, fid, nid_{DST})\}$
```

When a binder moves, it is deleted locally and created on the remote location. Also, a pointer to the remote implementation is defined. When a binder is copied, the local binder is not consumed and the implementation location of the copy is updated. Finally, when the binder is updated, the implementation location is changed by replacing binders that still point to the local implementation (it is assumed that nid_{IMP} and nid_{DST} differ).

Both local and remote service assignments can be moved and translated from local to remote and vice-versa.

```
$MOVE\_SRVASSIGN(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{inp}(\mathsf{srvAssign}, ?sid, fid, ?data, ?nid_{SRC}))$
      $\{\textbf{out}(\mathsf{srvAssign}, sid, fid, data, nid_{SRC})@nid_{DST}\}$

$MOVE\_REMOTESRVASSIGN(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{inp}(\mathsf{remoteSrvAssign}, ?\mathsf{sid}, fid, ?\mathsf{data}, ?\mathsf{nid}_{SRC}))$
      $\{\textbf{out}(\mathsf{remoteSrvAssign}, sid, fid, data, nid_{SRC})@nid_{DST}\}$

$TRANStoREM\_SRVASSIGN(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{inp}(\mathsf{srvAssign}, ?sid, fid, ?data, ?nid_{SRC}))$
      $\{\textbf{out}(\mathsf{remoteSrvAssign}, sid, fid, data, nid_{SRC})@nid_{DST}\}$

$TRANStoLOC\_SRVASSIGN(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{inp}(\mathsf{remoteSrvAssign}, ?sid, fid, ?data, ?nid_{SRC}))$
      $\{\textbf{out}(\mathsf{srvAssign}, sid, fid, data, nid_{SRC})@nid_{DST}\}$
```

To prevent losing requests, local to remote translation is necessary to move offloading/migration requests which have not yet been served. Similarly, remote to local translation is used in a back action when offloading terminates. Also, implementations can be moved or copied.

```
$MOVE\_IMPLEMENTATION(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{inp}(\mathsf{srvImplem}, ?sid, fid, ?Proc))$
      $\{\textbf{out}(\mathsf{srvImplem}, sid, fid, Proc)@nid_{DST}\}$

$COPY\_IMPLEMENTATION(fid, nid_{DST}) =$
  $\textbf{while}\ (\textbf{readp}(\mathsf{srvImplem}, ?sid, fid, ?Proc))$
      $\{\textbf{out}(\mathsf{srvImplem}, sid, fid, Proc)@nid_{DST}\}$
```

Threads can only be moved or started in that forced termination is not allowed. In the following, threads handling macros are introduced.

```
$MOVE\_THREADS(fid, nid_{DST}) =$
  $\mathbf{while}\ (\mathbf{inp}(\mathsf{thread}, fid, ?tid))\{$
      $\mathbf{out}(\mathsf{moveThread}, tid, nid_{DST});$
      $\mathbf{out}(\mathsf{thread}, fid, tid)@nid_{DST}\}$

$START\_THREAD(sid, fid, data, nid_{SRC}) =$
  $\mathbf{read}(\mathsf{srvImpl}, sid, fid, ?Code);$
    $\mathbf{fresh}(tid);$
    $\mathbf{out}(\mathsf{thread}, fid, tid);$
    $\mathbf{eval}(Code(tid, data, nid_{SRC}, fid))$
```

When a copy request is received, the copy action handler *copies* all binders (by setting the remote NAM as the *fid* location) and implementations, and sends copy requests to the service and policy handler.

```
$CH =$
  $\mathbf{in}(\mathsf{copyReq}, ?fid, ?nid_{DST});$
    $COPY\_BINDER(fid, nid_{DST});$
    $COPY\_IMPLEMENTATION(fid, nid_{DST});$
    $\mathbf{out}(\mathsf{copySH}, fid, nid_{DST});$
    $\mathbf{out}(\mathsf{copyPH}, fid, nid_{DST});$
    $PMH$
```

Upon the arrival of a migrate request, the migrate action handler *moves* all binders (by setting the remote NAM as the *fid* location), implementations, service assignments and threads, and sends migrate requests to the service and policy handler.

```
$MiH =$
  $\mathbf{in}(\mathsf{migrateReq}, ?fid, ?nid_{DST});$
    $MOVE\_BINDER(fid, nid_{DST});$
    $MOVE\_IMPLEMENTATION(fid, nid_{DST});$
    $MOVE\_SRVASSIGN(fid, nid_{DST});$
    $MOVE\_THREADS(fid, nid_{DST});$
    $\mathbf{out}(\mathsf{migrateSH}, fid, nid_{DST});$
    $\mathbf{out}(\mathsf{migratePH}, fid, nid_{DST});$
    $PMH$

$OH =$
  $\mathbf{in}(\mathsf{offloadReq}, ?fid, ?nid_{DST});$
  $\mathbf{out}(\mathsf{offloaderNAM}, fid, \mathbf{self})@nid_{DST};$
  $UPDATE\_BINDER(fid, nid_{DST});$
  $MOVE\_IMPLEMENTATION(fid, nid_{DST});$
  $TRANStoREM\_SRVASSIGN(fid, nid_{DST});$
  $MOVE\_THREADS(fid, nid_{DST});$
  $\mathbf{out}(\mathsf{offloadSH}, fid, nid_{DST});$
  $\mathbf{out}(\mathsf{offloadPH}, fid, nid_{DST});$
  $PMH$
```

Upon the arrival of an offload request (offloadReq) the handler adds a tuple tagged as offloaderNAM to notify the remote NAM nid_{DST} that its owner (**self**) is the functional module *fid*'s offloader. Then, it *updates* the binder for each service in *fid* to notify the new location nid_{DST} of the module. Afterwards, the handler *moves* the implementation of each service (*i.e.*, the code associated with each service in the functional module) to nid_{DST}. Each service assignment which has not been served yet is translated into a remote request and sent to nid_{DST}. Threads are *moved* to nid_{DST} by creating a moveThread tuple for each thread identifier tid. Each thread then accordingly reacts to the mobility request. Offloading requests are locally sent to the service and policy handler by using offloadSH and offloadPH requests, respectively, and by indicating the destination NAM nid_{DST}. Section 3.3.3 describes SH_{fid} reactions to requests. Finally, the control returns to *PMH* where either a policy or a mobility request is handled.

Upon the arrival of a back request, the *BH* back action handler sends a go request with destination **self** to the remote NAM.

```
$BH =$
  $\mathbf{in}(\mathsf{backReq}, ?fid, ?nid_{DST});$
    $\mathbf{out}(\mathsf{goReq}, fid, \mathsf{self})@nid_{DST};$
    $PMH$
```

The go action handler presents two possible behaviors. The first one is enacted on the remote NAM and retrieves the identity of the offloader NAM upon the arrival of a go request. Then, it sends a notification (goNotification) to the offloader NAM with the new location (NAM_2), so that such a NAM updates service bindings accordingly, and *moves* implementation and threads to the new location. If the new destination is the offloader itself, then the action is indeed a back action, which simply sends back requests to the service and policy handler and translates remote service assignments to local ones. Otherwise, it performs three sub-steps: (i) it informs NAM_2 of the offloader identity using the offloaderNAM tuple, (ii) it sends go requests to the service and policy handler, and (iii) it moves remote (not yet served) service assignments. The second behavior of *GH* is enacted on the local NAM and reacts to goNotification messages by updating service binders to point to the new NAM (possibly, **self**).

```
$GH =$
  $\mathbf{in}(\mathsf{goReq}, ?\mathit{fid}, ?\mathit{nid}_{DST});$
    $\mathbf{in}(\mathsf{offloaderNAM}, \mathit{fid}, ?\mathit{nid}_{OFF});$
    $\mathbf{out}(\mathsf{goNotification}, \mathsf{self}, \mathit{fid}, \mathit{nid}_{DST})@\mathit{nid}_{OFF};$
    $\mathbf{in}(\mathsf{goACK}, \mathit{fid});$
    $\mathit{MOVE\_IMPLEMENTATION}(\mathit{fid}, \mathit{nid}_{DST});$
    $\mathit{MOVE\_THREADS}(\mathit{fid}, \mathit{nid}_{DST});$
    $\mathbf{if}\ (\mathit{nid}_{DST} == \mathit{nid}_{OFF})$
      $\mathbf{then}\ \{$
        $\mathbf{out}(\mathsf{backSH}, \mathit{fid}, \mathit{nid}_{DST});$
        $\mathbf{out}(\mathsf{backPH}, \mathit{fid}, \mathit{nid}_{DST});$
        $\mathit{TRANStoLOC\_SRVASSIGN}(\mathit{fid}, \mathit{nid}_{DST})$
      $\}\ \mathbf{else}\ \{$
        $\mathbf{out}(\mathsf{offloaderNAM}, \mathit{fid}, \mathit{nid}_{OFF})@\mathit{nid}_{DST}$
        $\mathbf{out}(\mathsf{goSH}, \mathit{fid}, \mathit{nid}_{DST});$
        $\mathbf{out}(\mathsf{goPH}, \mathit{fid}, \mathit{nid}_{DST});$
        $\mathit{MOVE\_REMOTESRVASSIGN}(\mathit{fid}, \mathit{nid}_{DST});$
      $\};$
    $\mathit{PMH}$
  $+\ \mathbf{in}(\mathsf{goNotification}, ?\mathit{nid}_{SRC}, ?\mathit{fid}, ?\mathit{nid}_{DST});$
    $\mathit{UPDATE\_BINDER}(\mathit{fid}, \mathit{nid}_{DST});$
    $\mathbf{out}(\mathsf{goACK}, \mathit{fid})@\mathit{nid}_{SRC};$
    $\mathit{PMH}$
```

Chapter 4

NAM4J Middleware

This chapter presents the extension of NAM4J, carried out as part of the PhD activities. NAM4J is an open source Java middleware which has been specifically developed to implement NAM-based autonomic systems[1]. The middleware answers to the need for a tool integrating context-awareness and location-awareness in *Global Ambient Intelligence (GAmI)* applications [38]. Contextual information characterizes the situational user context and involves spatial, temporal and environmental parameters. However, context sources are manifold and acquired data must comply with a model that enables aggregation and reasoning activities. Also, providing a publish/-subscribe mechanism that triggers services, based on events, is crucial. The NAM4J middleware has been developed to cope with all such issues.

A layered stack, showing the role of NAM4J in a networked system, is presented by Fig. 4.1 [41]. NAM4J runs on top of the operating system on a physical or virtual device, and exploits an appropriate Java Virtual Machine (JVM). It also efficiently runs on Android devices, thus allowing for the implementation of MCC systems which involve both stationary and mobile devices.

[1]`https://github.com/dsg-unipr/nam4j`

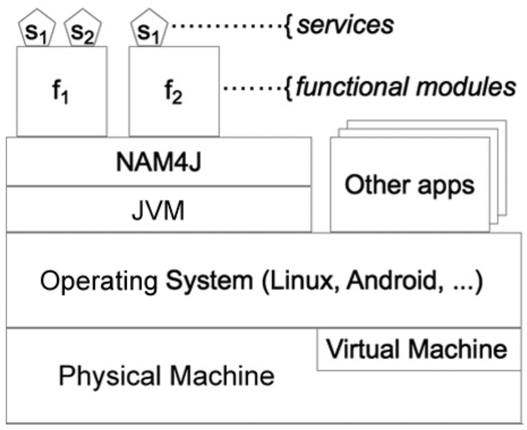
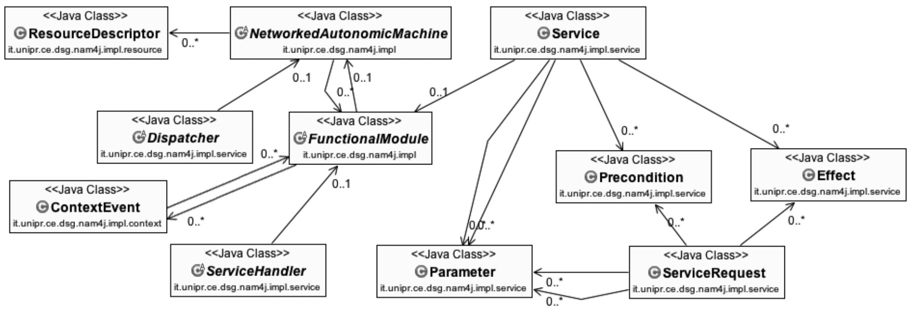


Figure 4.1: NAM4J layer stack.

4.1 NAM4 architecture

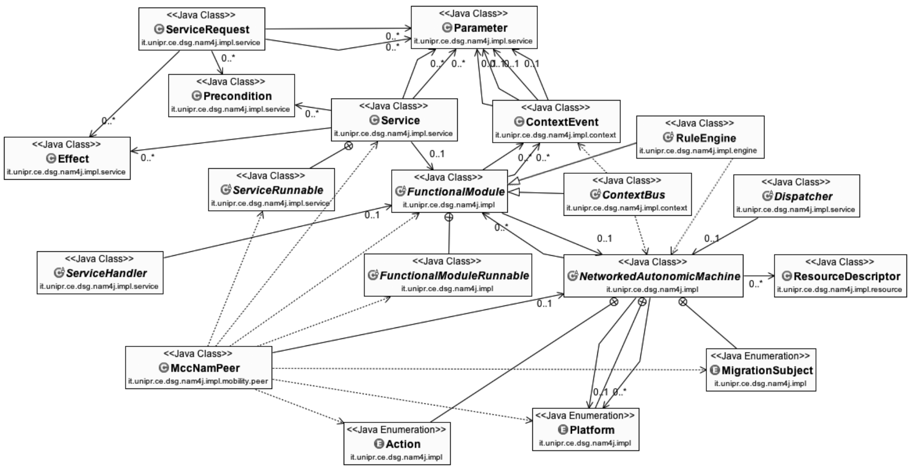
Before this PhD activity, NAM4J only included a reduced set of core classes which allowed to implement NAM entities with limited capabilities, as illustrated by Fig. 4.2. As part of the PhD activity, the code of such classes has been implemented and the middleware has been enriched with several components with the purpose of managing large scale autonomic distributed environments. Fig. 4.3 presents the class diagram of the current NAM4J version. In particular, the NamPeer class (together with related classes) allows for the implementation of P2P networks of NAM nodes. Also, the context bus and the rule engine functional modules have been added to provide, respectively, publish/subscribe capabilities and autonomic policies management, as described in sections 4.3 and 4.5. Moreover, NAM mobility actions have been implemented to enable the migration of code and execution state (Fig. 4.4). The offloading process involves the decomposition in chunks of items to be migrated (*i.e.*, functional modules, services, dependencies, execution state and resources) and follows the protocol described in section 4.2 (for the sake of readability, the full NAM4J class diagram has been split in figures 4.3 and 4.4 joined by the MccNamPeer class).

The development of NAM4J-based applications requires the extension of the NetworkedAutonomicMachine class. Each NAM node has an instance of such a class,



<<Java Class>> ResourceDescriptor it.unipr.ce.dsg.nam4j.impl.resource
<<Java Class>> NetworkedAutonomicMachine it.unipr.ce.dsg.nam4j.impl
<<Java Class>> Service it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Dispatcher it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> FunctionalModule it.unipr.ce.dsg.nam4j.impl
<<Java Class>> Precondition it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Effect it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> ContextEvent it.unipr.ce.dsg.nam4j.impl.context
<<Java Class>> ServiceHandler it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Parameter it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> ServiceRequest it.unipr.ce.dsg.nam4j.impl.service

Figure 4.2: NAM4J UML class diagram before the PhD activity.



<<Java Class>> ServiceRequest it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Parameter it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Precondition it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> Service it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> ContextEvent it.unipr.ce.dsg.nam4j.impl.context
<<Java Class>> RuleEngine it.unipr.ce.dsg.nam4j.impl.engine
<<Java Class>> Effect it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> ServiceRunnable it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> FunctionalModule it.unipr.ce.dsg.nam4j.impl
<<Java Class>> ContextBus it.unipr.ce.dsg.nam4j.impl.context
<<Java Class>> Dispatcher it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> ServiceHandler it.unipr.ce.dsg.nam4j.impl.service
<<Java Class>> FunctionalModuleRunnable it.unipr.ce.dsg.nam4j.impl
<<Java Class>> NetworkedAutonomicMachine it.unipr.ce.dsg.nam4j.impl
<<Java Class>> ResourceDescriptor it.unipr.ce.dsg.nam4j.impl.resource
<<Java Class>> MccNamPeer it.unipr.ce.dsg.nam4j.impl.mobility.peer
<<Java Enumeration>> MigrationSubject it.unipr.ce.dsg.nam4j.impl
<<Java Enumeration>> Platform it.unipr.ce.dsg.nam4j.impl
<<Java Enumeration>> Action it.unipr.ce.dsg.nam4j.impl

Figure 4.3: NAM4J UML class diagram after the PhD activity.

which provides data structures and methods to manage activities characterizing the NAM (*e.g.*, loading specific functional modules and performing mobility actions). Functional modules are defined through the extension of the FunctionalModule class. Such modules are then linked to the main class of the system through the addFunctionalModule(FunctionalModule) method, provided by the NetworkedAutonomicMachine class. Services exposed by functional modules are defined through the extension of the Service class. Similarly to functional modules, which are linked to a class extending NetworkedAutonomicMachine, services are linked to functional mod-

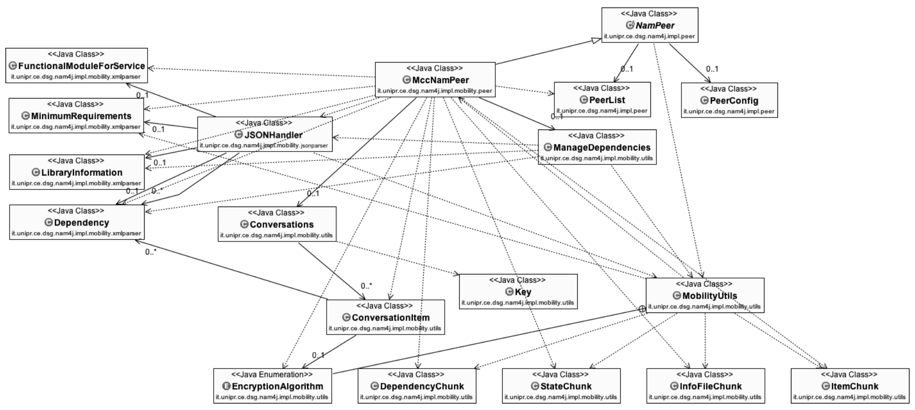
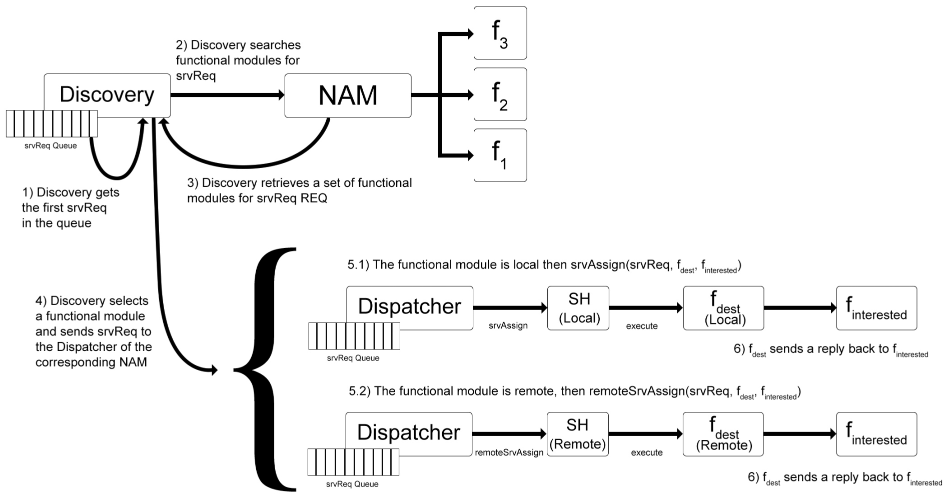

<<Java Class>>
NamPeer
it.unipr.ce.dsg.nam4j.impl.peer
<<Java Class>>
FunctionalModuleForService
it.unipr.ce.dsg.nam4j.impl.mobility.xmlparser
<<Java Class>>
MccNamPeer
it.unipr.ce.dsg.nam4j.impl.mobility.peer
<<Java Class>>
PeerList
it.unipr.ce.dsg.nam4j.impl.peer
<<Java Class>>
PeerConfig
it.unipr.ce.dsg.nam4j.impl.peer
<<Java Class>>
MinimumRequirements
it.unipr.ce.dsg.nam4j.impl.mobility.xmlparser
<<Java Class>>
JSONHandler
it.unipr.ce.dsg.nam4j.impl.mobility.jsonparser
<<Java Class>>
ManageDependencies
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
LibraryInformation
it.unipr.ce.dsg.nam4j.impl.mobility.xmlparser
<<Java Class>>
Dependency
it.unipr.ce.dsg.nam4j.impl.mobility.xmlparser
<<Java Class>>
Conversations
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
Key
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
MobilityUtils
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
ConversationItem
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Enumeration>>
EncryptionAlgorithm
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
DependencyChunk
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
StateChunk
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
InfoFileChunk
it.unipr.ce.dsg.nam4j.impl.mobility.utils
<<Java Class>>
ItemChunk
it.unipr.ce.dsg.nam4j.impl.mobility.utils
0..1
0..*

Figure 4.4: NAM4J UML mobility class diagram.

ules through the addConsumableService(Service) and addProvidedService(Service) methods provided by the FunctionalModule class. The latter class also provides the ServiceHandler nested class, which allows for the interaction between a functional module and its services. It is worth noting that each NAM can only access its own resources, while the interaction with resources of other NAMs happens through services, as described by Section 3.1.

For example, a NAM executing on a node provided with a number of sensors measuring different physical quantities may expose a service for each sensor. Services that provide data related to the same environment (*e.g.*, temperature, humidity, pressure) may be grouped by a single functional module, whose purpose is to describe the overall state of the environment. Interested nodes can access each sensor through the associated service.

To provide service interaction, we have included in NAM4J the IDispatcher interface, which must be implemented by classes characterized by different dispatching algorithms (according to the Strategy pattern [47]). We have implemented the dispatcher described in Section 3.3 (Dispatcher class), provided with a queue that handles incoming service requests, an algorithm that assigns service requests to local or remote functional modules, and a threadpool that handles incoming messages from remote NAMs. Fig. 4.5 [41] illustrates how service requests are managed.



Discovery
srvReq Queue
2) Discovery searches functional modules for srvReq
NAM
f3
f2
f1
1) Discovery gets the first srvReq in the queue
3) Discovery retrieves a set of functional modules for srvReq REQ
4) Discovery selects a functional module and sends srvReq to the Dispatcher of the corresponding NAM
5.1) The functional module is local then srvAssign(srvReq, fdest, finterested)
Dispatcher
srvAssign
SH (Local)
execute
fdest (Local)
finterested
6) fdest sends a reply back to finterested
5.2) The functional module is remote, then remoteSrvAssign(srvReq, fdest, finterested)
Dispatcher
remoteSrvAssign
SH (Remote)
execute
fdest (Remote)
finterested
6) fdest sends a reply back to finterested

Figure 4.5: Operations of the basic Dispatcher provided by NAM4J.

It is assumed that a discovery module is able to find NAMs that can serve a request — either the local one, or remote ones. The discovery module selects a NAM among the ones that offer a functional module capable of dealing with the service request and inserts such a request in the queue associated to its dispatcher. Two cases of service request assignments are possible. In the first case (Fig. 4.5, step 5.1), the destination functional module is local. Thus, the Dispatcher calls the srvAssign() method on the ServiceHandler of the chosen functional module. The ServiceHandler then calls the execute() method on the destination functional module, passing the description of the requested service. The destination functional module has a threadpool to handle incoming execution requests. When a requested execution completes, a reply is sent to the interested functional module (*i.e.*, the one that initially sent the service request to the Dispatcher). In the second case (Fig. 4.5, step 5.2), the destination functional module has been previously offloaded to a remote NAM. Thus, the service request is sent to a remote ServiceHandler, which calls the execute() method on the destination functional module and includes the description of the requested service. As for the first case, when the requested execution completes, a reply is sent to the interested functional module.

4.2 Mobility support for MCC applications

This section describes the implementation of NAM mobility actions in NAM4J. As illustrated by Fig. 4.4, all classes implementing code mobility features are grouped into the it.unipr.ce.dsg.nam4j.impl.mobility package.

In a NAM network, peers have a list of utils.ConversationItem objects, each representing a mobility action occurring between peers. Such an object includes a unique random key and is used to store data regarding the action, such as its type (COPY, MIGRATE, OFFLOAD, GO or BACK string), the id of the item to be transferred and the contact address and platform of the partner peer. Such an object is necessary in that each action is composed of asynchronous messages exchanged based on a protocol that allows nodes to request all missing dependencies for the migrated item, the item itself and possibly its state. Thus, each peer can hold multiple conversations (*i.e.*, manage multiple mobility actions) with different peers in parallel. In the following, code examples related to COPY and MIGRATE mobility actions are reported.

In order for migration to work properly, each functional module and service is described by a JSON file whose name is the identifier of the item itself. Such a file presents the structure illustrated by listing 4.1.

Listing 4.1: Example of a JSON item descriptor.

```
{
 "lib": {
  "info": {
   "id": "TestService",
   "version": "1.0",
   "type": "SERVICE",
   "main": "it.unipr.ce.dsg.examples.migration.TestService"
  },
  "functional_module": {
   "functional_module_version": "1.0",
   "functional_module_id": "TestFunctionalModule"
  },
  "dependencies": [
   {
    "dependency_type": "LIB",
        "dependency_version": "1.7.1",
        "dependency_id": "gson"
      },
      {
        "dependency_type": "FM",
        "dependency_version": "1.0",
        "dependency_id": "FunctionalModuleDependance"
      }
    ],
    "files": [
      { "file_id": "file.txt" }
    ],
    "requirements": {
      "cores": "2",
      "clock": "1024",
      "ram": "1024",
      "storage": "1",
      "network": "false",
      "location": "false",
      "camera": "false"
    }
  }
}
```

The JSON root element is lib, including the following attributes.

- info contains general information regarding the item:
  - id represents the unique identifier of the item, and must be equal to the names of the item itself and of the JSON descriptor file (*e.g.*, the binary for an item having id TestService must be named *TestService.jar*, and its JSON descriptor file must be named *TestService.json*);
  - version is the version of the item (peers can request a specific one);
  - type is either FM or SERVICE; it is used by the library to manage the received item (*i.e.*, a service is bound to a functional module, so the system tries to perform such a binding before the service execution starts);
  - main is the name of the item's main class, and the client calls its constructor when receives the item.

- functional_module attribute is only required for JSON files describing Services. Such an element describes the version and identifier of the functional module to which the Service is bound.
- dependencies attribute lists items necessary for the execution of the migrated item. In the example of listing 4.1, TestService requires Gson version 1.7.1 or newer, and FunctionalModuleDependance version 1.0 or newer. If the client does not have such items, related files are sent before the item itself. Upon receiving files, the client adds each to the class path, if the item is a functional module or a library. The item type is specified by the dependency_type element which takes one of the following values: INFO, RESOURCE, LIB, FM. NAM services do not depend on other services, therefore the associated type is not defined. If a dependency is a functional module, it is not necessary to specify its associated JSON descriptor file as another dependency, in that the system handles it when the type is specified as FM.
- files is a list of files required for the execution.
- requirements is the list of system specifications required for the execution.
  - cores is the minimum number of processing units;
  - clock is the minimum processor clock (Hertz);
  - ram is the minimum amount of RAM (Megabytes);
  - storage is the minimum storage space (Megabytes);
  - network is set to true if the item requires network access, false otherwise;
  - location is set to true if the item requires location information (*i.e.*, it requires the availability of a location sensor), false otherwise;
  - camera is set to true if the item requires the availability of a camera, false otherwise.

Developers can add custom requirements and edit jsonparser.JSONHandler class, which parses the JSON file, and decideWhetherToAcceptMobilityRequest() method of utils.MobilityUtils class, which checks for minimum requirements availability.

NAM4J supports state migration through serialization. We added an abstract Runnable class to FunctionalModule and Service classes, respectively named FunctionalModuleRunnable and ServiceRunnable, whose code is executed by the item and whose state can be migrated. As Java threads cannot be serialized, state migration requires serializing such a runnable, migrating it, deserializing it, and creating a new thread to execute its code. The provided runnables include four methods to manage the described cycle (start, stop, suspend and resume) and two methods the programmer overrides (if necessary) to save and retrieve non-serializable Java objects (saveState and restoreState). When a peer wants to start the execution of a service, the start method is called. When the peer wants instead to migrate the execution to another peer, saveState and suspend methods are called to send all required items as well as the execution state, as illustrated by listing 4.2.

Listing 4.2: Starting and suspending the execution of a service.

```java
Service s = new MyService();

// Functional module to which the service has to be bound
MyFunctionalModule fm = new MyFunctionalModule(myNam);

s.setFunctionalModule(fm);
s.getServiceRunnable().start();

// The service executes on it own thread

s.getServiceRunnable().saveState();
s.getServiceRunnable().suspend();
```

The receiving peer adds all items to the class path, binds the service to the proper functional module and calls ServiceRunnable's restoreState and resume methods. Such methods manage state deserialization, create a new thread and recover the execution. When the execution is resumed, the runnable's run method starts from the beginning, thus state recovery is successful only if developers keep outside of the run method the definitions of all variables whose state has to be saved. Also, developers can properly override the runnable's restoreState method to set variables' values, as illustrated by listing 4.3.

Listing 4.3: Resuming a service execution.

```
Service s = (Service) deserializedState;
JSONHandler infoFmHandler = MobilityUtils.parseJSONFile(functionalModuleId,
    this.nam);
String fmCompleteMainClassName =
    infoFmHandler.getLibraryInformation().getMainClass();

FunctionalModule fm = MobilityUtils.addServiceToFm(s, functionalModuleId,
    this.nam, fmCompleteMainClassName);
s.setFunctionalModule(fm);

// Resuming the service execution
s.getServiceRunnable().restoreState();
s.getServiceRunnable().resume();
```

The copy action happens when a peer (client) asks for an item (a functional module or a service) to another peer in the network (server). No state is transferred and the server keeps migrated items. The communication between two peers is based on messages (classes in it.unipr.ce.dsg.nam4j.impl.messages package) exchanged through the Sip2Peer middleware^2. The client generates a random key, creates a new ConversationItem object and stores it in its conversations list. Then, it sends a RequestCopyMessage message, which includes the key, to the server. The latter creates a corresponding ConversationItem object, sets the received key as the conversation identifier and stores the object in its conversations list. Therefore, both peers use the same key to identify the conversation they are having. The server checks for the availability of the requested item and, if it is not available, it answers with an ItemNotAvailableMessage message, otherwise it sends a RequestItemAnswerMessage message which includes the list of dependencies for the item. Dependencies are libraries, files and minimum system specification required for the execution. The server sets the RequestItemAnswerMessage message content by parsing the JSON descriptor file associated to the requested item. When the client receives an answer message, it checks for missing dependencies and, if any is needed, it sends a RequestDependenciesMessage message including a list to the server. Otherwise, it checks for the availability of the JSON descriptor file for the item. If such a file is not available, the client answers with an AllDependenciesAreAvailableMessage message, otherwise it sends an InfoFileIsAvailableMessage message to the server.

When the server receives a RequestDependenciesMessage message, it sends all requested items to the client. Each file is divided in chunks, and each chunk is sent as a DependencyChunk message. When the client has received all parts for a certain file, chunks get ordered, the file is rebuilt from them and is added to the class path at runtime. When all dependencies have been received, the client checks for the availability of the JSON file describing the item. If it is not available, the client answers with an AllDependenciesAreAvailableMessage message, otherwise it sends an InfoFileIsAvailableMessage message to the server.

When the server receives an AllDependenciesAreAvailableMessage message, it sends the item descriptor file as a set of InfoFileChunk messages. When the client has received such a file, it answers with an InfoFileIsAvailableMessage message. Finally, when the server receives such a message, it sends the requested item as a set of ItemChunk messages. When all chunks have been received, the client builds the file, adds it to the class path, retrieves its type from the descriptor file and answers with an ItemIsAvailableMessage message to notify the end of the conversation. Both peers then remove the associated ConversationItem from the list of ongoing conversations and store the identifiers of the item and the partner in a proper list, which is an attribute of NetworkedAutonomicMachine class. Such a list allows keeping track of every received and sent item and of the associated partners. If the item is a service, the name of the functional module to which it has to be bound is retrieved from the descriptor file, and such a module's execution starts. Then the service is added to the class path, bound to the functional module and started as well. If instead the item is a functional module, its execution starts after it has been added to the class path.

The migrate mobility action happens when a NAM is executing the code of a functional module or a service, and decides to suspend it and delegate another node in the network for it. The message exchange is the same of the copy action, but when the receiving peer has obtained the item, it waits for the state so that the execution re-

[2]https://github.com/dsg-unipr/sip2peer

sumes from where it was suspended. The sender provides the state as the serialization of the related runnable when it receives an ItemIsAvailableMessage message. As for every other exchanged entity, the state is divided in chunks sent through StateChunk messages.

The offload, go and back mobility actions are managed in a similar fashion to the described copy and migrate. Actions rely on the fact that all NAMs keep track of every peer from which an item was received, and to which an item was sent. Therefore, when an item is moved, it is possible to update bindings and redirect requests to the new responsible for it.

4.3 Context Bus

We developed a specific functional module, described by the ContextBus abstract class, which is particularly useful in collaborative applications in that, if replicated on each peer as illustrated by Fig. 4.6 [48], it instantiates a virtual shared channel allowing nodes to notify and get notified about context events. Such events may represent changes that happen in the environment and may influence the behavior of the distributed system. To provide a highly scalable publish/subscribe mechanism, the context bus is based on Sip2Peer middleware[3]and allows for three operations: publish an event occurrence, subscribe to get notified when a certain event occurs and unsubscribe from a previous subscription.

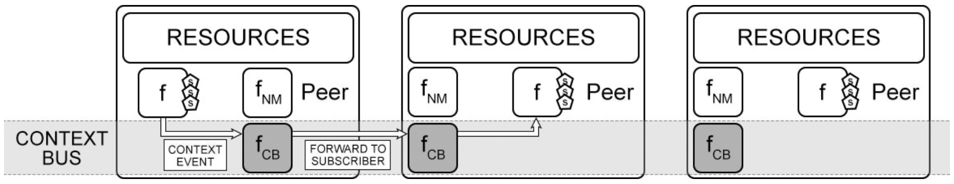


Figure 4.6: Context Bus involving three peers.

In collaborative applications that benefit from the context bus presence, nodes are distributed in a P2P network whose topology is managed by proper functional

[3]https://github.com/dsg-unipr/sip2peer

modules.We have developed modules which allow the deployment of Chord [49], Kademlia [50], full mesh and random graph topologies. To join the network, a new node contacts a bootstrap peer, supposed to be known, which replies with a list of active nodes, according to the network architecture. Every node in a NAM4J-based network can be the bootstrap and, therefore, the seamless replacement of nodes that play such a role is possible in case of failures or if problems occur.

In a full mesh topology, the bootstrap communicates to new nodes the list of all connected peers. Thus, each node has a full knowledge of the network. Since all possible pairs of nodes are connected, full mesh networks are characterized by high-rate connection. Moreover, such networks are extremely robust in that, since the number of links in a N-peers network is $N(N-1)/2$, until a node is not isolated, there exist at least one path connecting it to every other node in the network. However, the square law hinders scalability, thus, such a topology is advisable only for small networks. Fig. 4.7 represents the sequence diagram for network joining, and event publishing, subscribing and unsubscribing in full mesh networks.

The scalability issue is solved by partial mesh networks where each node has a partial knowledge of the network itself, whose topology depends on the used network manager functional module. The number of links in such a topology is between $N-1$ and $N(N-1)/2$. The robustness of partial mesh networks decreases proportionally to the number of links. However, such a topology is more advisable for large-scale networks. In NAM4J-based partial mesh networks, the initial peer list of a new node only includes peers communicated by the bootstrap. Since such a number is greatly smaller than the whole number of peers, the system forwards requests for a predetermined number of hops. The forwarding process causes continuous updates to peer lists in that, when a node receives a subscription request, it adds the subscriber to its peer list. Fig. 4.8 illustrates the sequence diagram for network joining and event publishing, subscribing and unsubscribing, in partial mesh networks.

When peers become interested in a certain type of event, they notify every active peer they know through the context bus. Figure 4.9 illustrates such a behavior in a partial mesh network. Every node that receives a subscription request stores it and forwards it to its peer list. Such a behavior is reiterated until the request reaches

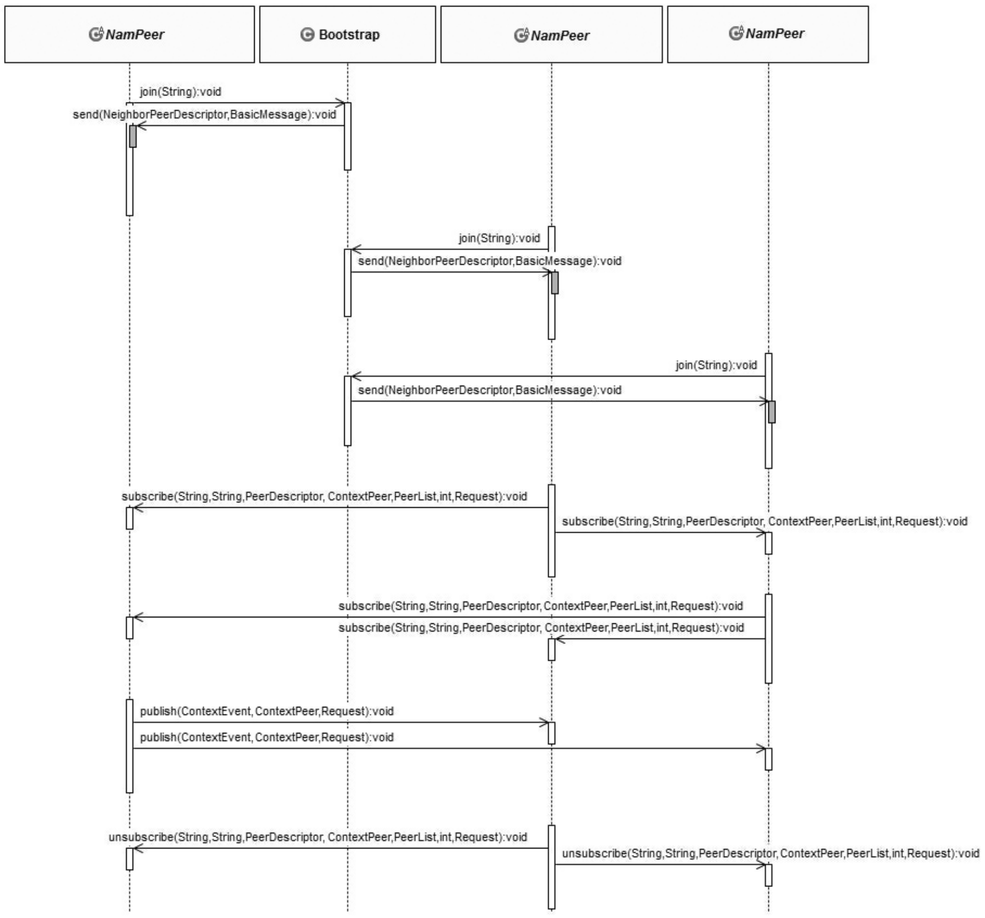

NamPeer
Bootstrap
NamPeer
NamPeer
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
publish(ContextEvent, ContextPeer,Request):void
publish(ContextEvent, ContextPeer,Request):void
unsubscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
unsubscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void

Figure 4.7: Sequence diagram for network joining, and event publishing, subscribing and unsubscribing in full mesh networks.

a predetermined number of hops from the subscriber. Then, when a node detects an event occurrence, the context event is notified to all active peers that previously subscribed to such type of event, as presented by Figure 4.10 [41]. In the example, nodes $N4$ and $N6$ generate an event and notify $N3$ which forwards the information to $N2$. To increase anonymity, when a node receives a subscription request, it stores the requestor id and forwards it using its own id. For example, in Figure 4.9 [41], $N2$

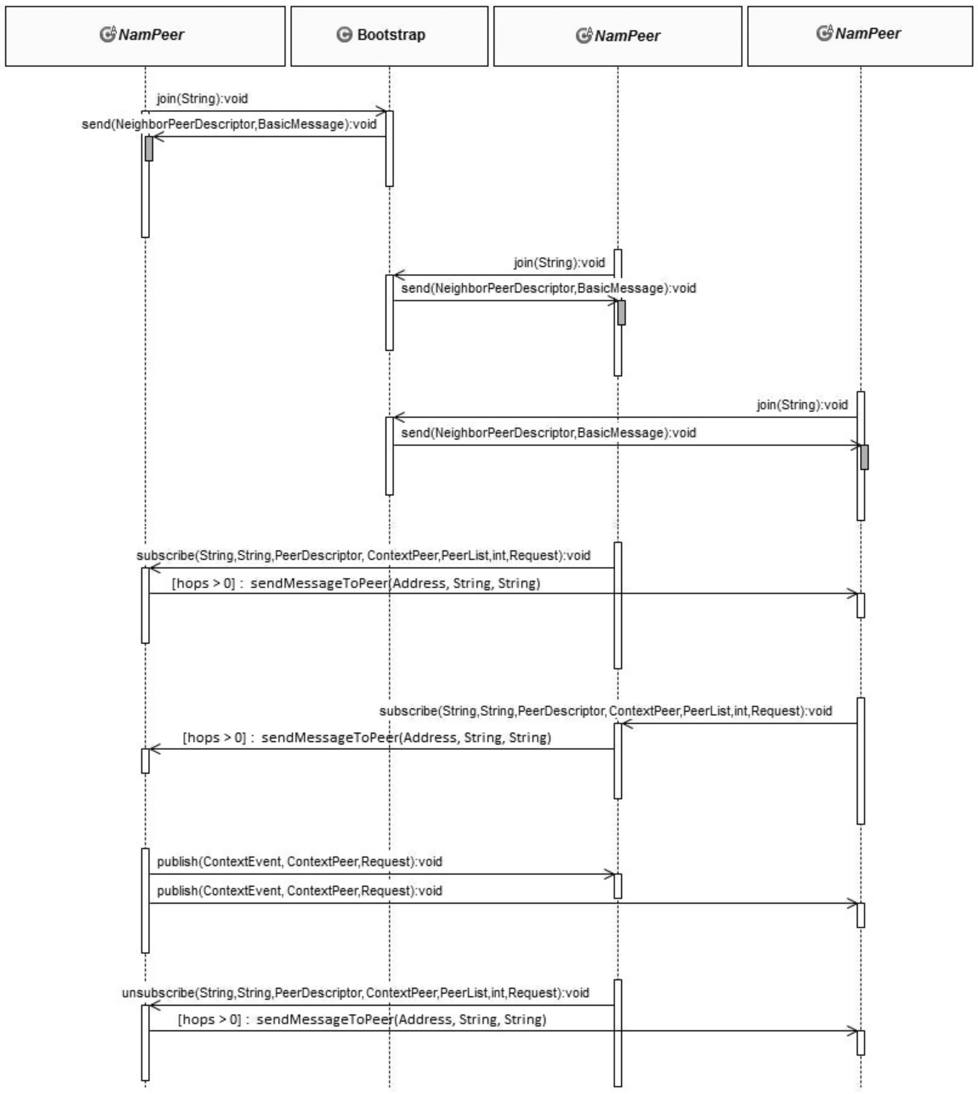

NamPeer
Bootstrap
NamPeer
NamPeer
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
join(String):void
send(NeighborPeerDescriptor,BasicMessage):void
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
[hops > 0] : sendMessageToPeer(Address, String, String)
subscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
[hops > 0] : sendMessageToPeer(Address, String, String)
publish(ContextEvent, ContextPeer,Request):void
publish(ContextEvent, ContextPeer,Request):void
unsubscribe(String,String,PeerDescriptor, ContextPeer,PeerList,int,Request):void
[hops > 0] : sendMessageToPeer(Address, String, String)

Figure 4.8: Sequence diagram for network joining, and event publishing, subscribing and unsubscribing in partial mesh networks.

sends a subscription request to $N1$, subscription(N2, e), which is forwarded to $N6$ as subscription(N1, e). Therefore, when a node receives a request, it is not aware of the

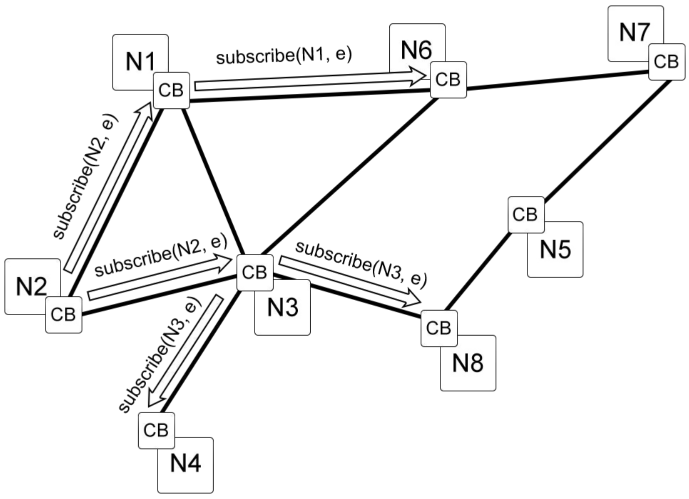


Figure 4.9: Propagation of subscription messages.

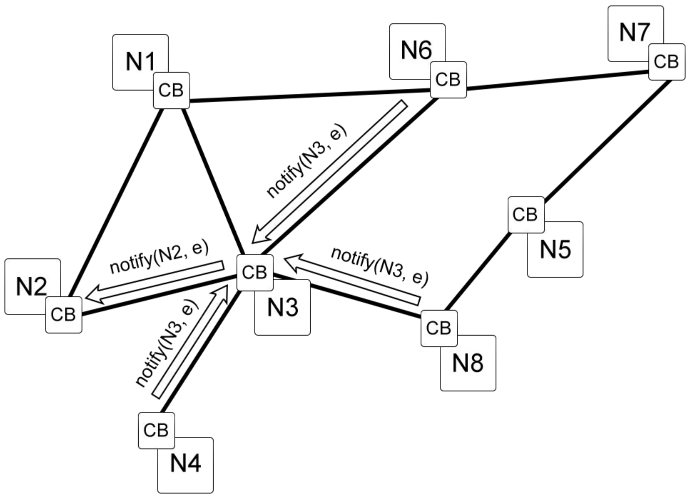


Figure 4.10: Propagation of notification messages.

original node. Thus, when events occur, each node notifies the peer from which the request was received, until the notification reaches the actual origin.

4.4 DARTSense

NAM4J includes trust and reputation management mechanisms to make the middleware suitable for applications in which such aspects are of great interest. An example is represented by collaborative applications, such as participatory sensing, where privacy is one of the main concerns. Tagging sensed data with location and time can involuntarily reveal personal information. On the other hand, users need to trust received information, and a completely anonymous system can hinder data reliability. To cope with both aspects, among existing solutions to address privacy and trust issues, Anonymous Reputation and Trust Sensing (ARTSense) [51] is the most advanced in that it provides trust and reputation management while maintaining user anonymity.

ARTSense is a sound framework, introducing an innovative mechanism for peer reputation updating. The reputation level of every participant is determined considering only her/his behavior. Participants do not have control over the process, can provide their level without revealing their identity and cannot lie. The ARTSense framework addresses the privacy-reliability tradeoff by means of a centralized, server-based architecture and adds the concepts of *trust* and *reputation* to participatory sensing applications. Given a sensing report r, trust $T(r)$ is the probability that such a report is correct when is received by the server. Also, given a participant P_i, reputation $R(P_i)$ is the probability that past reports produced by P_i are correct. The server also generates and updates a reputation estimate for each participant, denoted as reputation level $\hat{R}(P_i)$.

Sensing reports consist of two parts, namely the payload and the provenance. While the former represents sensing data, the latter details the origin of the report and is divided in *user-provenance*, to describe the user while preserving anonymity, and *contextual-provenance*, to depict the sensing environment. In addition, reports include a Reputation Certificate (RC) signed and granted by the server, which contains the sender's reputation level. The server uses the RC to assess the trust of received reports.

Trust is computed considering the location, time, sensing mode and means of transportation, in that such parameters influence the quality of acquired information. The quality of a report regarding location L_t, produced in location L_s, gets more accurate as distance $|L_s - L_t|$ decreases. Location concealing techniques obfuscate the actual position such that the contextual-provenance includes an area centered in L_s and having diameter D_c in which the user is approximately located. Such an area allows for computing the *location distance factor* as:

$$\Theta = e^{D_c \cdot \alpha} \cdot \left(1 - e^{-|L_s - L_t| \cdot \alpha}\right) \tag{4.1}$$

where α represents the influence of such a factor on trust. Time is another critical factor in that the validity of reports decreases over time. The validity of a report produced at time T_t, evaluated at time T_s is higher if the difference $|T_s - T_t|$ is small (*i.e.*, the report is recent). If S_c is the duration of the time interval used for concealing, the *time gap factor* is defined as:

$$\Omega = e^{-S_c \cdot \beta} \cdot \left(1 - e^{-|T_s - T_t| \cdot \beta}\right) \tag{4.2}$$

where β represents the influence of such a factor on trust. The *base trust* of a sensing report can be computed as:

$$T_b(r) = \hat{R}(P_i) \cdot (1 - \Theta_r) \cdot (1 - \Omega_r) \cdot \lambda_r \cdot \mu_r \tag{4.3}$$

where λ and μ are milieu factor weights, respectively for report generation and exchange, developers define based on the application. The base trust represents an important factor related to a single report for a task. However, for many applications a node receives more reports related to the same task. Such reports can agree or contrast and their *similarity value* $S(r, r')$ takes values in set $[-1, +1]$ where -1 indicates complete conflict and $+1$ complete agreement. When all similarity values for each report pairs in a C set have been computed, it is possible to compute the *similarity factor* as:

$$\Delta_r = \frac{\sum_{r, r' \in C_r, r \neq r'} S(r, r')}{|C_r| - 1} \cdot e^{-\frac{1}{|C_r|}} \cdot \gamma \tag{4.4}$$

where $|C_r|$ is the number of reports for a given task and γ represents the influence of such a factor on trust. Negative values of Δ_r indicate that a given task has more conflictual reports than complying ones, and vice-versa. $T_b(r)$ and Δ_r are used to compute the *final trust* value as:

$$T_f(r) = T_b(r) \cdot (1 + \Delta_r) \tag{4.5}$$

The ARTSense architecture is centralized in that a server hosts a database to store reputation levels. Participants willing to execute a task register on the server by sending a task registration request which includes the participant id and the task id (TID). The server stores the information in its database, gets the participant's reputation value based on the most recent $R(P_i)$ value and creates and sign the RC certificates pair. The latter includes the reputation $\hat{R}(P_i)$ and the TID. Certificate RC_0 is added to the user-provenance and RC_i is used by the node to compute a blinded id as:

$$BID = RC_i \cdot b^e \, (modN) \tag{4.6}$$

where b is a random number different for each report. After the report has been generated, it is sent to the server which computes a *reputation feedback level* f_r and a *reputation feedback coupon* (RFC) for the sender as:

$$RFC = [BID]_{Kspriv} \Big| \Big[(fr)_{Kspub} \Big| RC_0 \Big]_{Kspriv} \tag{4.7}$$

In the RFC, f_r is encrypted with the server public key so that a participant receiving the coupon does not know if the feedback is positive or negative. The only action it enacts, is the replacement of the first part (*i.e.*, the *BID*) with RC_i certificate encrypted with its private key to get the *Unblinded* RFC (URFC). When the server receives a URFC validates it, extracts P_i and f_r, and updates the corresponding reputation value in its database.

The centralized approach has well-known drawbacks, such as the presence of a single point of failure and scalability issues. To deal with such problems, we have designed and implemented a distributed version of the framework, denoted as DARTSense, where no central server is required, as reputation values are stored and updated by participants in a subjective fashion (*i.e.*, every participant has its own view of other participants and does not need to share it).

4.4.1 Distributed reputation management

Every participant computes the reputation of information providers based on the difference N between the sum of positive matchings n_+ and the sum of negative matchings n_- (Eq. 4.8), when comparing received events which concern the same task and location. N is used to compute reputation values using the *sigmoid* function (Eq. 4.8), illustrated by Fig. 4.11, which takes values in $]0,1[$ and is characterized by a trend which is particularly suitable for our purposes, in that it makes it difficult to reach very low and very high levels of reputation.

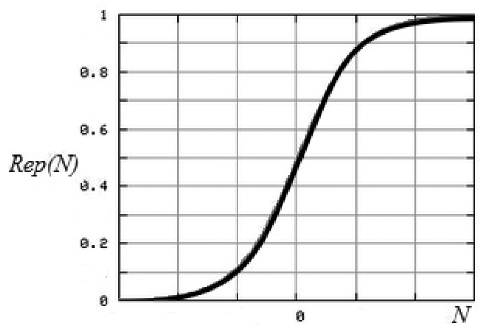


Figure 4.11: The sigmoid function used to compute reputation values.

$$N = \sum n_+ - \sum n_- \tag{4.8}$$

$$Rep(N) = \frac{1}{1+e^{-N}} \tag{4.9}$$

Every peer keeps trace of the identifiers of good and bad peers (*i.e.*, peers that respectively provide true and false information). Such sets are identified by a threshold R_{th}, whose value must be lower than default reputation for the i-th peer R_i. Moreover, reputation values and related timestamps are also stored by every peer. If a peer sends a subscription request, and is not in any list yet, it is added to the good peers one, with initial reputation R_i. If a peer notifies a context event, its identifier is consequently added to one of the lists, based on the reputation update process described in the following. Every participant also has the Valid Context Events (VCE) set, to store all context events that are still valid. Such a set is essential for the reputation update process, in that N is computed by taking into account only valid events. The reputation is updated when a context event is received, and the VCE set includes a number of events greater than a specified threshold n_{th}, or when a context event is generated, and the VCE set includes at least one event received from another participant.

Peers add received events to the VCE set and, when its size is at least n_{th} and a new event is received, the system iterates through all VCE events. For each event, the system checks if the provider is already in either the good or the bad peers list. We defined three system behaviors to deal with all possible situations. In any of them, n_+ and n_- are increased by a different value ($\Delta n < \Delta n' < \Delta n''$) according to the importance of the current situation.

When the VCE size is at least n_{th}, the node browses the set, gets the provider's id of each event and checks if it is already in the good or bad peers list. If not, the provider's reputation is initialized to R_i and the event is compared to other events in VCE of the same type and related to the same location. If events agree, n_+ is incremented by Δn, otherwise n_- is. Such a process is repeated for each event in VCE and, in the end, reputation values are computed by using the sigmoid function. Such values, and the current time, are then added to the good or bad peers list based on R_{th} threshold.

When the system chooses an event from VCE whose provider is already in the good or bad peers list, it retrieves the provider's current reputation value and the time it was last updated. If the event is later than the provider's last reputation update time, it is compared to all valid events of the same type and related to the same location. In case of an agreement, n_+ is incremented by $\Delta n'$, otherwise n_- is. When the event has been compared to all events in VCE, the reputation value of the participant is updated, as well as the last update time.

When an event is received from a participant which previously provided an event in VCE, such a set must be updated. The new event is added to the set, while the previous one provided by the same participant is removed. Let be t the time it had been generated. For every event in VCE, t is compared to the time t_i in which the i-th event had been generated. If $t > t_i$, the system behavior is described by Fig. 4.12, otherwise the event is skipped. Then, the validity of all events in VCE is checked. To do so, the time ΔT passed since their generation is compared to the $timestamp$ specified when they were generated. If $\Delta T \geq timestamp$, the event is no more valid and is therefore removed from VCE. Fig. 4.13 describes the behavior of the system concerning events which are still valid. When all events have been checked, the reputation of the participant that provided the last received event, as well as the reputation of all valid events providers, is updated and stored in good or bad peer list, based on R_{th} threshold.

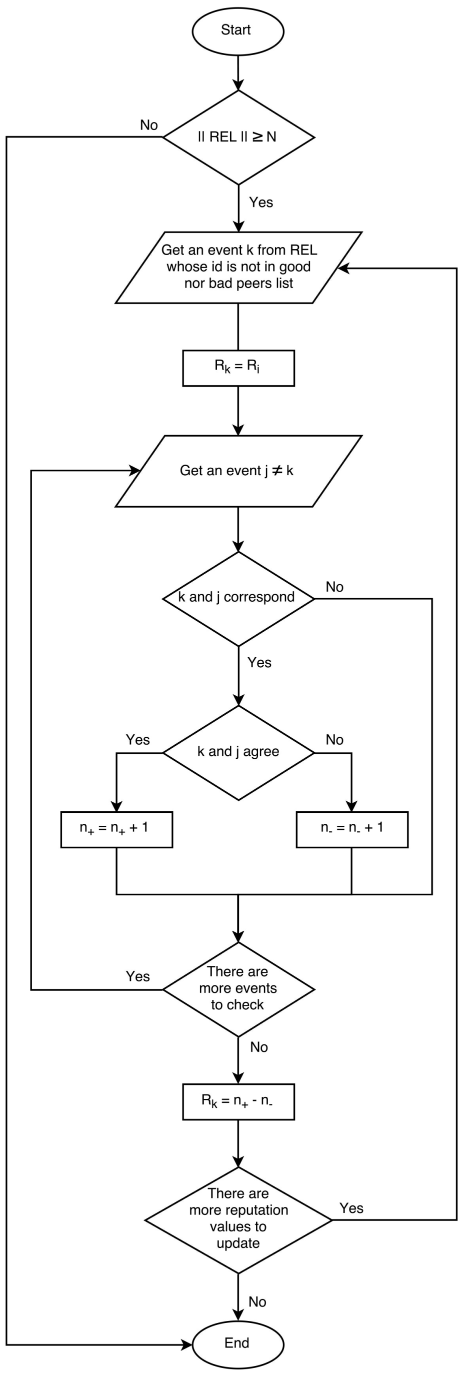


Figure 4.12: Flow chart of the reputation value updating process, when the number of received events is greater than n_{th}.

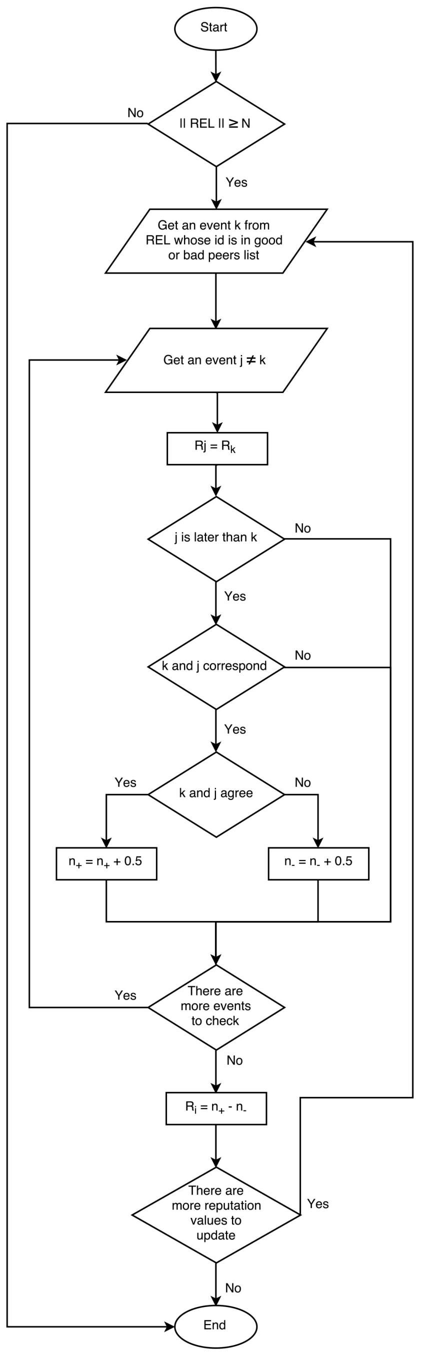


Figure 4.13: Flow chart of the reputation value updating process, when the updating has already been performed at least once.

When a participant subscribes to a certain event type, it starts receiving related event notifications. If the participant also senses the environment by itself, to publish in the network events of the same type, it is able to assess the validity of previously received events by comparing them to its own sensed data. In fact, since the participant knows the latter is correct, it is able to update the reputation of participants which provided related events. In case of an agreement, n_+ is increased by $\Delta n''$, otherwise n_- is. In the end, the reputation of all related event providers is updated and stored in good or bad peers list, based on R_{th} threshold. In the described situation the increment is higher than in the other cases since the node knows for certain (unless its sensors do not perform well) if previously received events are valid or not. Fig. 4.14 illustrates such a system behavior.

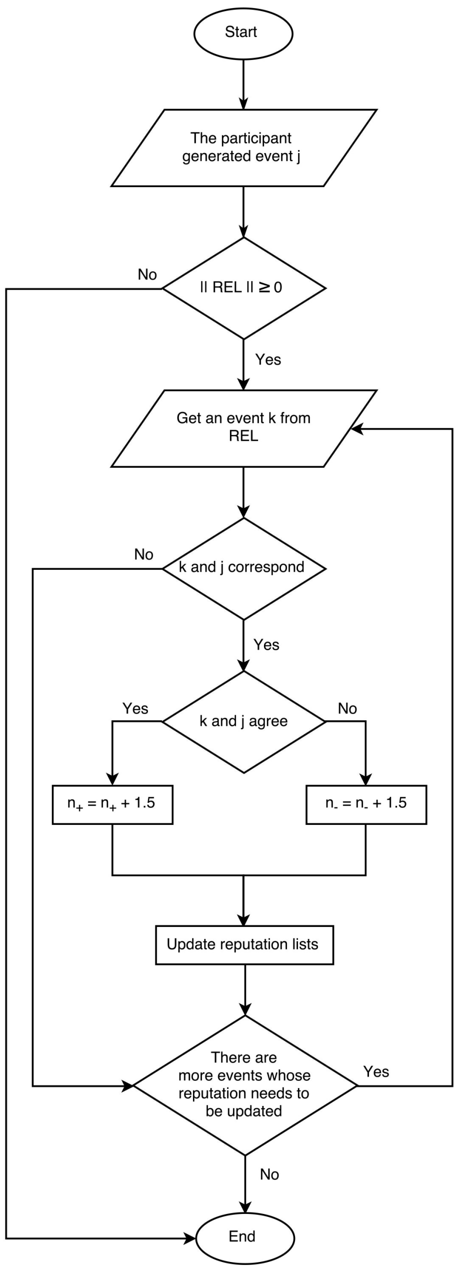


Figure 4.14: Flow chart implementing reputation values updates when an event is generated and others of the same type and location had been received by other participants.

4.5 Rule engine

Autonomic MCC requires the definition of policies allowing the system to determine when offloading is convenient. To this purpose, we provided NAM4J with a rule engine which uses ECA rules. Inference engines are systems which use rules to produce results or to change the system state. The inclusion of a rule engine brings several benefits.

- The separation between data and rule sets simplifies the implementation, maintenance and update process of an application. Since the whole decisional logic resides in rules, changes do not affect it.
- Traditional rules management based on *if ... then* statement makes the application evaluate conditions even when there is no actual need. Thus, using a rule engine improves performance and scalability.
- Using a rule set allows the definition of a knowledge base which represents a *single-point-of-truth* for management policies.
- A rule engine allows the simple representation of solutions to complex problems and is generally more easily understandable with respect to traditional code.

NAM4J's rule engine is a functional module defined by the RuleEngine abstract class, and is based on JBoss Drools[4] component, a software allowing the design, implementation and maintenance of customizable rule sets. Drools is a business rule management system (BRMS) integrating a rule engine based on forward and backward chaining inference. Such a production rule system uses an enhanced implementation of the Rete algorithm [52], supports **JSR-94** standard for the construction and maintenance of business policies, and includes the following components.

- *Guvnor* is a centralized repository for Drools knowledge bases, allowing the management of rules, functions and processes (rules related to a specific knowl-

[4]http://www.drools.org/

edge base can be easily configured and extended through a web-based interface).

- *Expert* is the API core implementing the rule engine which performs reasoning. It allows the management of rules groups and supports decision tables based on Excel (XLS) and CSV files.

- *Fusion* provides for complex event processing (CEP) in which the identification of an event in the cloud (*i.e.*, the amount of information concerning all events in the knowledge base) depends on complex patterns such as abstraction, correlation, belonging and hierarchy relationships, and processes related to events.

- *Flow* allows the management of the business workflow through the rule engine, by grouping rules into flows.

- *Planner* allows the optimization of automated planning.

The implementation of NAM4J's rule engine is based on the Expert component and runs also on Android devices.

Drools has a rule language, which supports natural and domain specific languages via expanders which allow its adaptation to specific problem domains. A rule file (DRL) is typically a file with a .drl extension which can contain multiple rules, queries and functions, as well as resource declarations like imports, globals and attributes assigned to or used by rules and queries.

Drools is composed of two parts: *Authoring* and *Runtime*. The *Authoring* is the process of defining a DRL file including all rules which are fed to a parser. The role of the parser is to check the syntax correctness and produce a structure which describes rules in a proper manner. Such a structure is used by a package builder which generates the necessary code. The *Runtime* component is able to instantiate one or more working memories at each time instant. The working memory is a core component of Drools in that it is the space hosting *facts*, which represent data used by the rule engine. Facts are implemented as JavaBean classes (*i.e.*, serializable classes that encapsulate several objects into a single object, have a zero-argument constructor and allow to access properties using getter and setter methods) asserted into the working memory, where they can be modified or removed. When facts are asserted, one or more rules are true and scheduled for execution. The described method is named *forward chaining* and is illustrated by Fig. 4.15[5].

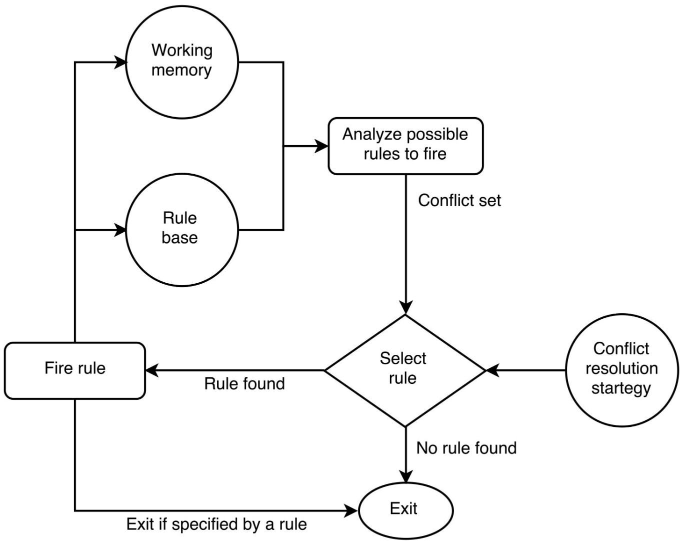

Working memory
Rule base
Analyze possible rules to fire
Conflict set
Select rule
Conflict resolution startegy
Fire rule
Rule found
No rule found
Exit
Exit if specified by a rule

Figure 4.15: Drools forward chaining execution method.

In the following, two simple rules related to device energy are described. One of the conditions used by such rules is based on the energy balance formula proposed by Kumar *et al.* [53], which expresses the offloading energy cost (Eq. 4.10).

$$c(B,C,F) = \frac{C}{M} \times (W_m - \frac{W_i}{F}) - W_{off} \times \frac{D}{B} \tag{4.10}$$

being C the job size in terms of number of instructions, M the job execution rate (instructions per second) on the mobile device, W_m the average power consumed to

[5]`http://www.mastertheboss.com/jboss-jbpm/drools/jboss-drools-tutorial`.

execute jobs on the mobile device, W_i the average power consumed by the mobile device when idle, F the (nominal or estimated at runtime) speedup of the Cloud, W_{off} the average power consumed by the job transfer (*i.e.*, moving data between the mobile device and the Cloud, mostly), D the amount of transferred data, B the bandwidth that is available to the mobile device. If $c(B,C,F) > 0$, the energy cost of local execution is higher than the offloading one.

The first sample rule – *Cloud* – enacts the execution migration to a remote cloud and is activated when at least one of the following conditions is satisfied.

- The battery level is greater than a specified threshold corresponding to the 30% of the full charge.
- The device is recharging.
- The energy balance formula (Eq. 4.10) result is positive.

The first and second conditions ensure that the device has enough battery to perform the offloading and receive results, while the third assesses the offloading convenience. Vice versa, the second sample rule – *Local* – which enacts local execution, is activated when none of the previous conditions is satisfied.

Data checked by the rule engine to perform inference can be included in a class defined as presented by listing 4.4. Methods of the class are then referenced in the DRL file which includes the two described rules. Listing 4.5 presents the content of such a file.

Listing 4.4: Example of a Java class holding data used to perform inference.

```
public static class Decision {
  public static final int MIN_LEVEL = 30;
  public static final int CHARGING = 0;
  private int batteryLevel;
  private int batteryStatus;
  private double energyBalanceFormulaResult;

  public int getBatteryLevel () {
    return this.batteryLevel;
  }

  public void setBatteryLevel(int batteryLevel) {
    this.batteryLevel = batteryLevel;
  }

  public int getBatteryStatus() {
    return this.batteryStatus;
  }

  public void setBatteryStatus(int batteryStatus) {
    this.batteryStatus = batteryStatus;
  }

  public double getEnergyBalanceFormulaResult() {
    return this.energyBalanceFormulaResult;
  }

  public void setEnergyBalanceFormulaResult(double
      energyBalanceFormulaResult) {
    this.energyBalanceFormulaResult = energyBalanceFormulaResult;
  }
}
```

Listing 4.5: DRL file example.

```
rule "Local"
  when
    Decision(getBatteryLevel() < Decision.MIN_LEVEL && getBatteryStatus()
        != Decision.CHARGING && getEnergyBalanceFormulaResult() < 0)
  then
    // Perform local execution
end

rule "Cloud"
  when
    Decision(getBatteryLevel() >= Decision.MIN_LEVEL || getBatteryStatus()
        == Decision.CHARGING || getEnergyBalanceFormulaResult() > 0)
  then
    // Perform cloud execution
end
```

Listing 4.6 presents the code used to pass rules to the engine. The get() and getKieClasspathContainer() methods of KieContainer class load the knowledge base, while newKieSession method of KieSession class creates a new session to feed rules to the engine and execute them. The parameter of the method is the object of the class which contains data to be analyzed, and fireAllRules() method starts the execution.

Listing 4.6: Code to pass rules to the engine as well as data to perform reasoning.

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getKieClasspathContainer();
KieSession kieSession = kieContainer.newKieSession("ksession-rules");
kieSession.insert(decision);
kieSession.fireAllRules();
```

Chapter 5 presents applications built on NAM4J with the purpose of evaluating the context bus, the reputation management mechanism, the rule engine, and to study the impact of autonomic policies on MCC scenarios.

Chapter 5

Analysis and evaluation

The purpose and the contribution of this chapter are twofold. On one hand, the first part introduces a modeling and simulation framework for the design and analysis of MCC systems provided with autonomic policies, in the context of MCC-oriented NAM extensions reported in Chapter 3. To this purpose, we developed a discrete event simulator for the evaluation of MCC systems. On the other hand, the second part of this chapter presents the application of NAM to two different scenarios. Firstly, we apply NAM principles to define a hybrid P2P/cloud approach where components and protocols are autonomically configured according to specific target goals, such as cost-effectiveness, reliability and availability. As an example, we show how the approach can be used to design robust collaborative storage systems based on an autonomic policy to decide how to distribute data chunks among peers and Cloud, according to cost minimization and data availability goals. Secondly, we define an autonomic architecture for urban participatory sensing (UPS) which bridges sensor networks and mobile systems to improve effectiveness and efficiency.

5.1 MCC quantitative analysis

The design of MCC systems is a challenging task, in that both the mobile device and the Cloud have to find energy-time tradeoffs, and the decisions taken by parts affect each other. The analysis of Abolfazli *et al.* [2], regarding the entire history of MCC, points out that all MCC models focus on mobile devices, and consider the Cloud as a system with unlimited resources [53, 54]. To contribute in filling this gap, we have proposed a modeling and simulation framework for the design and analysis of MCC systems, encompassing both their sides [55].

The MCC system is depicted as a Queueing Network (QN) [56] which includes two sub-networks: one for the Cloud and one for the mobile devices, as illustrated by Fig. 5.1. The Cloud model consists of a task Dispatcher, followed by a cluster of k VMs. The mobile device model includes a Local Engine (for local job execution) and an Offloader (for remote job migration). A generic mobile device is characterized by a job request rate λ, representing the rate at which the mobile user generates jobs for the device itself. A job j is defined by a set of independent tasks, whose size $N_t^{(j)}$ is a random variable (assumed to be uniform, for simplicity). Given a job, the mobile device executes its tasks either sequentially or in parallel, while the Cloud always executes them in parallel.

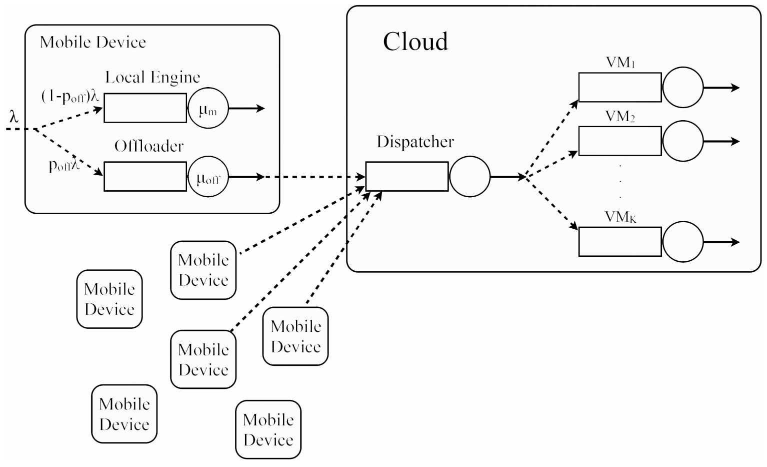


Figure 5.1: QN model of the MCC system.

We define the offloading probability as

$$p_{off} = P\left\{\bigcup_i cond_i\right\} \tag{5.1}$$

where $cond_i$ is the i-th offloading condition expressed by Eq. 4.10. If $c(B,C,F) > 0$, the energy cost of local execution is higher than the offloading one. In the following, for simplicity, we assume that every instruction requires one clock cycle. Thus, in Eq. 4.10, M is approximated with the clock frequency of the mobile device.

The average job execution time can be predicted by using Eq. 5.1:

$$\Delta t_e = p_{off}(\Delta t_{ec} + \Delta t_{off}) + (1 - p_{off})\Delta t_m \tag{5.2}$$

where Δt_{ec} is the average *job execution time on the Cloud*, Δt_{off} is the average time required by the offloading process, Δt_m is the average *job execution time on the mobile device* (p_{off} may depend, among others, on a constraint on the job execution time).

Once offloading has been decided, the tasks of the job are sent to the First-Come First-Served (FCFS) queue of the Offloader, to be moved to the Cloud. The job execution time on the Cloud Δt_{ec} is defined as the time interval between the arrival and the departure of the job in the Cloud. The *job service time on the Cloud* Δt_{sc}, instead, is defined as the time interval required by the Cloud to actually execute the job. The ratio $S_c = \Delta t_{ec}/\Delta t_{sc}$ is called *cloud slowdown.*

The QN representing the Cloud is inspired by the model proposed by Moschakis and Karatza [57], where the Dispatcher assigns tasks to VMs using one of the following algorithms:

- RANDOM: assign the task to a randomly selected VM;
- Shortest Queue First (SQF): assign the task to the VM with the shortest queue;
- Shortest Queue + Earliest Deadline First (SQEDF): find the VMs which guarantee to meet the job's deadline; among them, select the one which allows to meet the deadline earlier.

Each VM of the dynamic cluster uses one of the following job scheduling policies:

- First-Come First-Served (FCFS): tasks are served in the order they arrive into the queue of the VM;
- Shortest Job First (SJF): the waiting task with the smallest service time is selected to be processed next.

The model allows for the definitions of adaptive loops between the generic mobile device and the Cloud, in order to improve their interaction, including the offloading decision. For example, the mobile device periodically obtains updated speedup F and bandwidth B estimations, which are used to compute the energy cost tradeoff $c(B,C,F)$ and the job execution time on the Cloud Δt_e, to decide whether to offload or not the job itself. Meanwhile, the Cloud computes workload statistics, which are used to optimize the number of active VMs $k = k_{opt}$ and to provide refined speedup predictions.

As F strongly depends on the workload of the Cloud, each job offloading decision of the mobile device is based on an estimated — job-specific —F, explicitly provided by the Cloud. The mobile device computes the expected time to complete the jobs in its queue and adds it to the estimated local execution time of the target job. The resulting Δt_m is sent to the Cloud, together with a descriptor of the target job (*i.e.*, number of independent tasks, clock cycles of each task, etc.) and the Cloud estimates the time Δt_{ec} it would require to execute the target job. The estimated speedup is then computed as $F = \Delta t_m / \Delta t_{ec}$ and returned to the mobile device, to be used in the computation of c, by means of Eq. 4.10.

As the number of mobile devices N may randomly increase or decrease over time, the Cloud should be able to adapt the number of VMs k in order to preserve the already defined SLAs, and to provide the same offer to new clients. The number of active VMs may vary between the k_{min} and k_{max} values, and the Cloud periodically computes the updated value $k(t)$ at time t, by taking into account $k(t-1)$ and the ratio ϕ_{tc} between the average maximum time task waiting time and the maximum task execution time (*i.e.*, waiting time plus service time). The fraction f_{ct} of the number of completed tasks versus the number of arrived tasks is used to compute the k value:

$$k(t) = \begin{cases} k_{min} & \text{if } \lceil k(t-1) + \frac{\delta}{f_{ct}}(\lceil \phi_{tc} \rceil - s) \rceil < k_{kmin} \\ k_{max} & \text{if } f_{ct} = 0 \text{ or } \lceil k(t-1) + \frac{\delta}{f_{ct}}(\lceil \phi_{tc} \rceil - s) \rceil > k_{kmax} \\ \lceil k(t-1) + \frac{\delta}{f_{ct}}(\lceil \phi_{tc} \rceil - s) \rceil & \text{else} \end{cases}$$

where s is the setpoint and δ is a system parameter affecting the swing amplitude of $k(t)$ before the system reaches a stable configuration (ideally, the one where $\phi_{tc} = s$).

We have implemented a simulator of the MCC model based on the DEUS [58] general-purpose discrete event simulation environment. The simulator is highly modular and reusable in that it is possible to define different mobile device classes (each characterized by specific features), and the Offloader can be configured with the most appropriate characterization of the communication channel between the mobile device and the Cloud. It is possible to define any kind of policy for the Cloud's Dispatcher and the VMs, as well as the adaptive policy that updates the number of active VMs.

In the simulations, mobile devices are characterized by 1, 2 or 4 CPU cores (with distribution 30%, 60% and 10%, respectively) and the following parameters:

- battery capacity = 4500 mWh;
- $W_m = 0.6 \div 0.9$ W (0.5 W being the power consumed by the operating system and other processes executed in background with respect to the considered app);
- $W_i = 0.25$ W;
- $W_{off} = 0.7$ W;
- $M = 800$ or 1400 MHz (equally distributed);
- available RAM = 1 GB.

Table 5.1: Power demand of different mobile devices.

Model	WiFi transmission [W]	CPU-intensive [W]	Idle standby [W]
Openmoko Neo Freerunner [59]	0.7	0.67	0.25
T-Mobile G1 [60]	$0.6 \div 1.3$	n.a.	$0.1 \div 0.5$
HTC Magic [60]	$0.6 \div 1.3$	n.a.	$0.1 \div 0.5$
HTC Nexus One [61]	0.35	0.55	0.1
HTC Galaxy i7500 [62]	0.628	0.606	n.a.
HTC Nexus S [62]	0.455	0.886	n.a.
HTC HD2 [63]	0.817	n.a.	n.a.
HTC EVO [64]	n.a.	1.005	n.a.
Motorola ATRIX 4G [64]	n.a.	0.918	n.a.
Motorola Droid [64]	n.a.	0.944	n.a.

These values have been chosen by taking into account real devices (see Table 5.1).

We set the simulated Cloud parameters based on the OpenStack deployment in our Department, whose VMs are provided with a virtual processor Intel Core i7 9xx with 2 GHz clock. The nominal bandwidth between mobile device and Cloud is $B =$ 10 MB/s and the initial number of VMs is 10; $k_{min} = 5$, while $k_{max} = 120$.

Every mobile device generates jobs characterized by $N_t = 8$ as maximum degree of parallelism. Consider, for example, image processing jobs (face detection, license plate recognition), where each task corresponds to the processing of a picture acquired by the mobile device. Like Moschakis and Karatza [57], we consider *Low-Parallelism Jobs* (made of up to 4 tasks), with probability q, and *High-Parallelism Jobs*, (made of 4 to 8 tasks), with probability $1-q$, where q can assume values 0.25, 0.5 or 0.75. Every task is characterized by:

- number of instructions: $C/N_t = U(4 \cdot 10^8, 2 \cdot 10^9)$
- required RAM space: $R = U(1,5)$ MB
- input data size: $D = U(1,3)$ MB
- allowed completion time: 40 s

Table 5.2: Image Processing Statistics.

Width [Px]	Height [Px]	Size [KB]	# Faces	# Mobile Proc. Time [s]	Cloud Proc. Time [s]
1920	1080	50	3 + 1/2	4.02	2.93
1920	1080	188	3	4.7	2.78
1200	800	559	5	5.22	1.37
2048	1356	851	5	5.22	3.79
3504	2336	1900	3	19.48	11.55

The C range corresponds to 4 to 20 seconds of image processing, on the considered mobile devices and reflects the values we measured on a real device, illustrated in Table 5.2. The completion time for a job is given by the number of tasks that compose the job, multiplied by the allowed task completion time. Jobs are generated according to a Poisson process with rate λ, such that the inter-arrival time between jobs is 1280 s, on the average. This time period is the ideal worst-case service time for a job, obtained by multiplying N_t and the allowed completion time of every task.

Each job is assigned either to the Local Engine or to the Offloader of the mobile device. A job may also be dropped, if the mobile device estimates that the deadline of the job cannot be met neither locally nor remotely. $c(B,C,F)$ is computed using the estimated speedup of the Cloud. Thus, $c(B,C,F)$ may vary over time, even if the job statistics do not change. Also, every 300 [s] the Cloud computes the optimal k value and updates the number of active VMs.

To evaluate the performance of the MCC system, we measured the following indicators:

- offloading probability;
- % of completed jobs;
- average execution time for jobs Δt_e;
- % of completed tasks;
- % of deadline hits for tasks;

- average execution time for tasks $\Delta t'_e$;
- average cloud service time for tasks $\Delta t'_{sc}$;
- average cloud execution time for tasks $\Delta t'_{ec}$;
- average cloud slowdown for tasks S'_c;

versus N. We considered four N values, namely $\{100, 200, 300, 400\}$. Moreover, we measured the time variation of the number of active VMs and the time variation of the offloading probability, to analyze how they quickly converge in an interdependent way.

To validate the proposed model, we compared simulation results with experimental results obtained with the OpenStack-based private Cloud of our Department, using simulated client devices in both cases. On such a system, we deployed a RESTful web service, which accepts images as input and sends them to the VMs, where they are processed by an *OpenCV*–based face recognition program. We developed all components of the model (including a module that autonomically resizes the Cloud, in terms of active VMs, by means of the OpenStack API) and tuned parameters to match the simulator's workload. Mobile devices are simulated by threads which generate requests. Figure 5.2 reports validation results for the case of 100 simulated mobile devices.

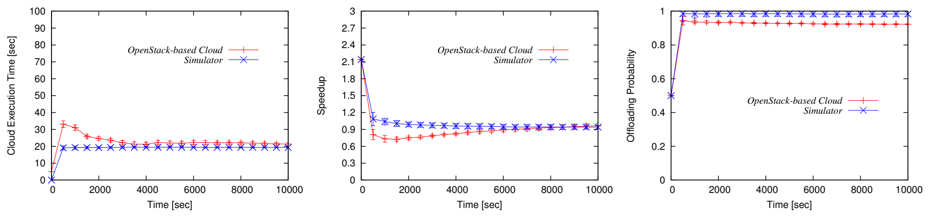


Figure 5.2: Average cloud execution time for tasks, estimated speedup for tasks and offloading probability versus time, when the number of devices is 100. The simulator is compared to the OpenStack-based private Cloud deployed in our Department.

Considering the SQF, SQEDF and RANDOM dispatching strategies, the average execution time for tasks is shown in Fig. 5.3, 5.5 and 5.7, respectively, for increasing number of mobile devices and different q values. Focusing on the Cloud only (graphs on the right), we observe that performance does not deteriorate when the total job rate increases and large jobs become the majority ($q = 0.75$).

Observing the SQF-related results, it appears that the SJF scheduling strategy does not outperform FCFS. This is due to the fact that tasks are quite homogeneous, as a consequence of the offloading policy adopted by mobile devices. For this reason, we report only the SJF results, for what concerns the RANDOM dispatching strategy. Instead, for the SQEDF strategy, we show only the FCFS results, as SQEDF assumes FCFS only.

The average cloud slowdown for tasks S_c' is illustrated in Figures 5.4, 5.6 and 5.8. With 100 devices, RANDOM appears to be the best dispatching strategy. However, when the number of devices increases, SQF becomes the best dispatching strategy. Again, we observe that SJF does not outperform FCFS, as a scheduling strategy.

Fig. 5.9 illustrates the percentage of deadline hits for tasks, considering the four dispatching and scheduling configurations described so far. These results confirm that SQF is the best dispatching strategy. Once more, we observe that SJF does not outperform FCFS, as a scheduling strategy. Moreover, we observe that SQEDF and SQF have the same performance. Thus, as SQF is more simple, it should be preferred.

Figure 5.10 illustrates the average execution time for jobs over the whole MCC system, considering the configurations of dispatching and scheduling described above. The SQF+FCFS, SQF+SJF and SQEDF+FCFS strategies are the best ones. The SQF+FCFS and SQF+SJF strategies are the best ones. Their worst-case value, which corresponds to $q = 0.25$ and $N = 400$ devices, is less than 60 s. The graphs show that such a value is lower than those of every other configuration.

The analysis of the simulation results shows that the best performance is obtained through the use of the SQF dispatching policy, in the Cloud. Figure 5.11 shows the time evolution of the number of active VMs and offloading probability for the SQF+SJF configuration, with $N = 100$ and $N = 200$ mobile devices and $q = 0.75$. When the initial number of VMs is $k(0) = 2$, the offloading probability decreases almost immediately. Thus, the Cloud has no reason to grow k. Also when $k(0) = 6$, with $N = 100$, we observe that $k(t)$ does not change. Setting $k(0) = 10$ does not im-

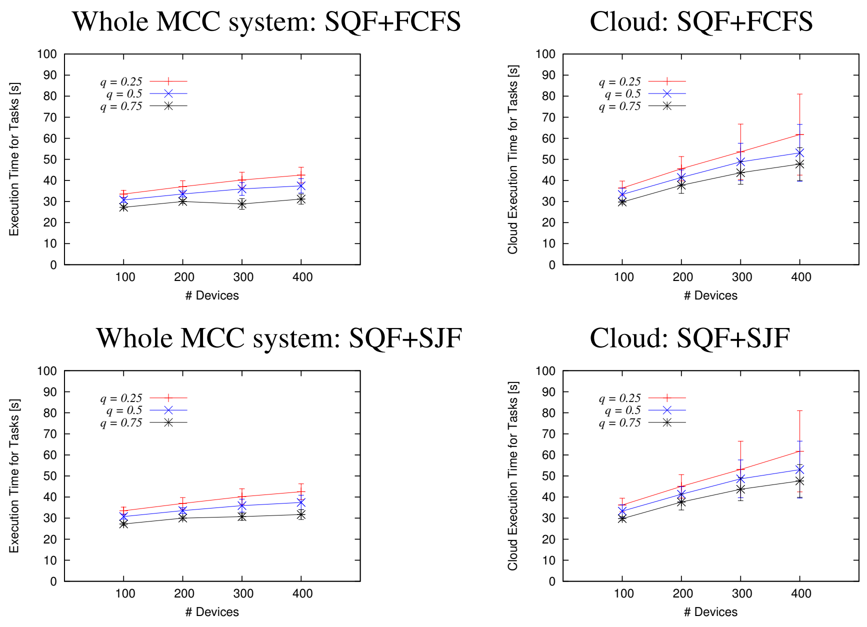


Figure 5.3: Average execution time for tasks, versus job rate, in the case of SQF dispatching and FCFS/SJF scheduling. The whole MCC system is compared to the Cloud alone.

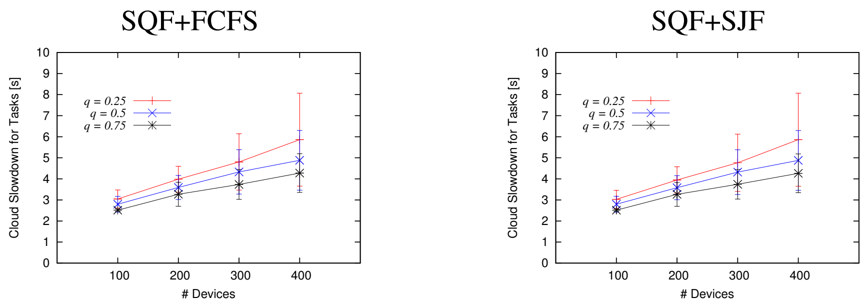
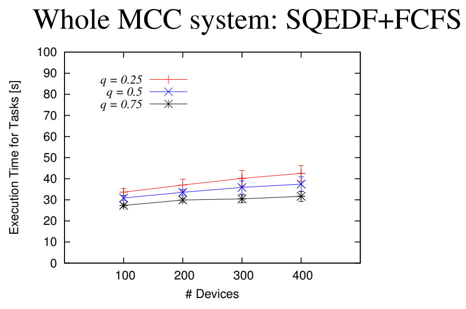


Figure 5.4: Average cloud slowdown for tasks, versus job rate, in the case of SQF dispatching and FCFS/SJF scheduling.

proves the performance of the Cloud. Indeed, the number of activated VMs slowly decreases towards the limit values (which is almost 6). To speed up the convergence,



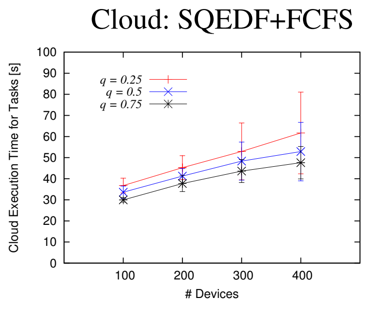


Figure 5.5: Average execution time for tasks, versus job rate, in the case of SQEDF dispatching and FCFS scheduling. The whole MCC system is compared to the Cloud alone.

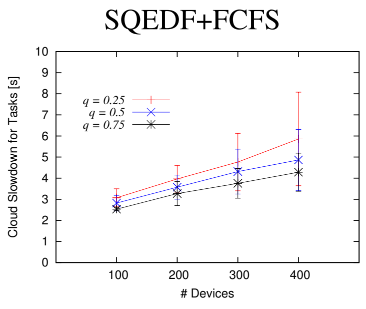


Figure 5.6: Average cloud slowdown for tasks, versus job rate, in the case of SQEDF dispatching and FCFS scheduling.

it would be necessary to use a larger δ value, but this would require a reduced VM (de)activation frequency. With $N = 200$, it is more evident that 6 VMs are not sufficient and 10 are too many. Finally, when the number of mobile devices increases from 100 to 200, with $k(0) = 6$ or higher, the reaction of the Cloud is sufficiently quick to maintain a stable offloading probability value.

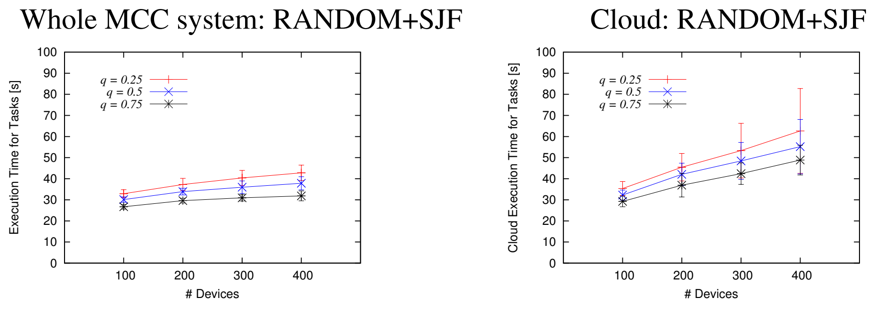


Figure 5.7: Average execution time for tasks, versus job rate, in the case of RANDOM dispatching and SJF scheduling. The whole MCC system is compared to the Cloud alone.

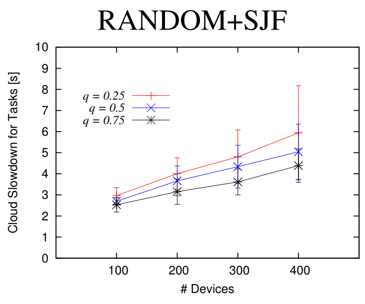


Figure 5.8: Average cloud slowdown for tasks, versus job rate, in the case of RANDOM dispatching and SJF scheduling.

5.2 P2P/Cloud integration

Efficient and cost-effective large scale distributed collaborative environments cannot be achieved by leveraging upon the Cloud alone, even if its superior availability makes it more appealing for web businesses, compared, for example, to the best effort philosophy of P2P [65, 48]. Cloud and P2P approaches are widely different in that, while the former is based on completely decentralized protocols, the latter follows the client/server paradigm to leverage on the presence of high performance data centers. Another difference is that P2P services are usually free and do no guarantee reliability nor high performance, while cloud services are usually fee-based and performance is

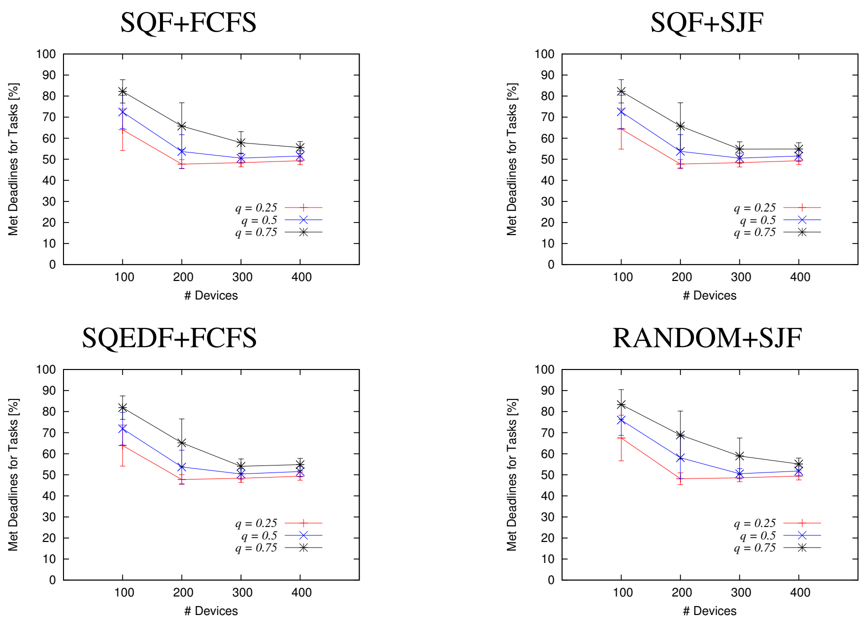


Figure 5.9: The percentage of deadline hits for tasks, considering the three dispatching strategies SQF, SQEDF and RANDOM, with FCFS/SJF scheduling.

contracted. However, a cloud infrastructure can benefit from the presence of a P2P network, as the responsibility of provided services can be shared among peers [66]. On the other hand, a P2P system can be augmented by the presence of clouds satisfying requirements beyond the reach of a P2P network [67]. Merging the Cloud and P2P paradigms brings together the advantages of both: high availability, provided by the Cloud presence, and low cost, by exploiting inexpensive peers resources.

During the PhD activity, we have defined a NAM-based hybrid P2P/Cloud approach where components and protocols are autonomically configured aiming to improve cost-effectiveness, reliability and availability [48]. The autonomic approach dynamically moves tasks from the Cloud to the P2P network, and vice versa, to meet both user-specific and collective goals. The P2P system is characterized as a set of heterogeneous peers having different capabilities and behaviors. To use a service provided by the Cloud, peers pay a fee, whose amount depends on how long

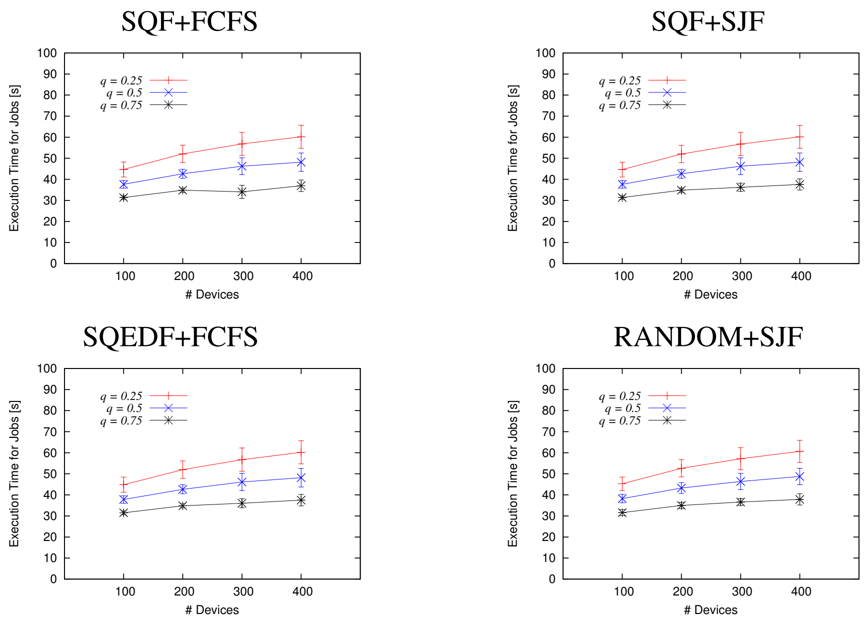
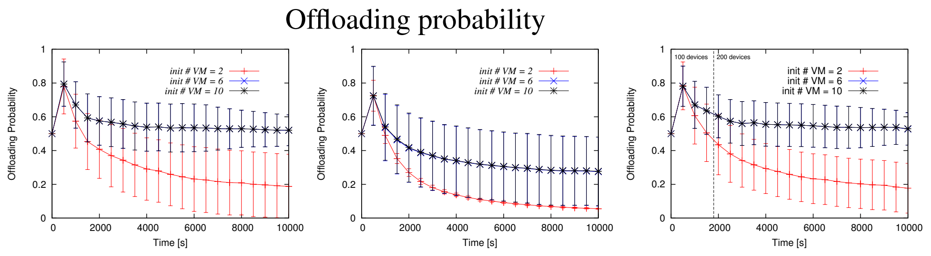


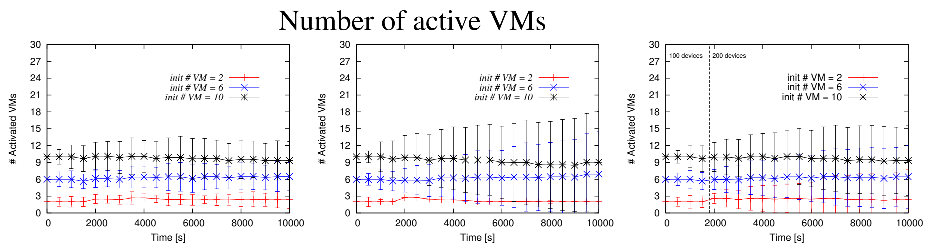
Figure 5.10: The average execution time for jobs over the whole MCC system, considering the three dispatching strategies SQF, SQEDF and RANDOM, with FCFS/SJF scheduling.

the service is used and on the contracted Quality of Service (QoS). Each peer has a limited knowledge of other peers and the P2P overlay network can be a structured or an unstructured mesh. In general, every NAM is provided with a functional module which manages connectivity and communication, denoted as f_{NM}. The context bus functional module, which operates on top of f_{NM}, is denoted as f_{CB}. As described in Section 4.3, the context bus allows peers to publish and consume context events. Moreover, the P2P/Cloud system allows to advertise and search for services. To improve scalability and fault-tolerance, it is possible to use a decentralized service bus, implemented as a functional module denoted as f_{SB} , which decouples service providers from service consumers (Fig. 5.12 [48]).

Peers are provided with autonomic policies that update the local perception of the P2P network, based on context events. Every peer builds and maintains its own



(a) $N = 100$ (b) $N = 200$ (c) $N = 100$ to 200



(d) $N = 100$ (e) $N = 200$ (f) $N = 100$ to 200

Figure 5.11: Time variation of the offloading probability and number of active VMs, considering different N values and $q = 0.75$, in the case of SQF dispatching and SJF scheduling.

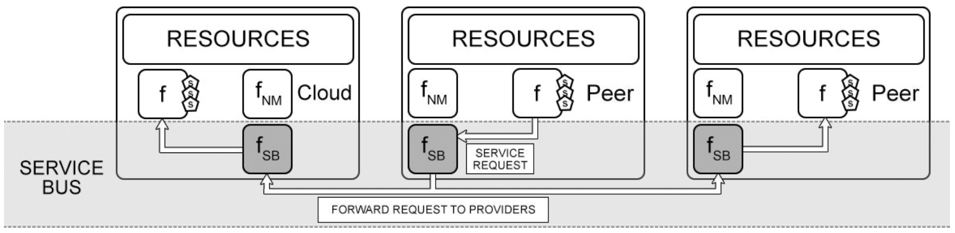


Figure 5.12: Service Bus involving two peers and one Cloud.

reputation ranking of other peers by considering the upload to download ratio and the average response time of the considered peer. Also, peers estimate the churn rate, *i.e.*, the frequency of node departures and arrivals. The local perception of the P2P network supports the peer in deciding whether to trust other peers or, instead, to exploit the Cloud, for the correct and efficient execution of its functional policies.

The defined approach has been applied to the design of a distributed collaborative storage system whose architecture is presented by Fig. 5.13 [48]. Every peer is provided with a Data Manager f_{DM}, which uses f_{SB} to store and retrieve data into the P2P/Cloud system, and f_{CB} to publish heartbeat information. f_{DM} maintains two tables: Table1 to index known peers, which are characterized by a reputation value, a reciprocity value and a presence flag, and Table2, to keep trace of where stored data are (either into the Cloud or into other peers). Periodically, the peers and the Cloud publish a heartbeat message to f_{CB}, to notify their presence to other interested peers, according to the context bus publish/subscribe scheme.

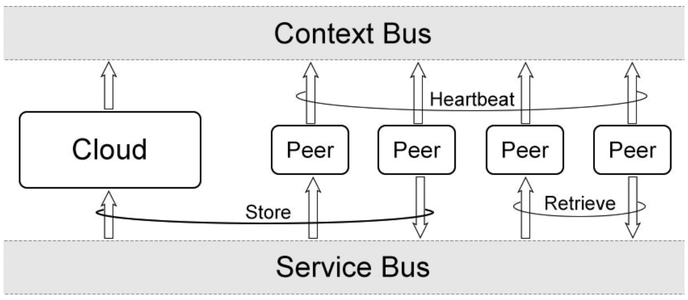


Figure 5.13: NAM-based distributed storage.

We assume that stored files are encoded by means of efficient maximum distance separable (MDS) erasure codes, *e.g.*, Reed-Solomon codes, parity-array codes, and low-density parity-check (LDPC) codes [68, 69]. With MDS coding, a k symbols information message is encoded into a message with $n > k$ symbols (thus, the coding rate is k/n). Later, users can retrieve the information by downloading a proper k-size subset of the encoded data chunks.

When peer p_x wants to store a file, and m_x highly reputed peers, among its neighbors (*i.e.*, the peers that p_x knows), are currently online. To save money, p_x can use them to store data chunks. The basic strategy is to uniformly distribute data chunks among highly reputed neighbors. Another approach is to order highly reputed neighbors by reputation and send n_i data chunks to the i-th ranked peer, where n_i is proportional to the *reputation* r_{ix} that p_x assigns to that peer.

The reputation is a subjective value, and, given two peers p_x and p_y, their reciprocal reputation values r_{xy} and r_{yx} depends on the percentage of fulfilled data chunk retrieval requests $r_1 = n_{fr}/n_r$ (dimensionless), and on the ratio between the average response time of the Cloud and the average response time of the peer $r_2 = \min\{1, rt_c/rt_p\}$ (dimensionless). Both r_1 and r_2 are in $[0,1]$ and, in general, we set $r \in [0,1]$. For example, r could be a linear combination of r_1 and r_2 ($r = w_1 r_1 + w_2 r_2$, where w_1 and w_2 are tunable weights). High reputation means that r is above a given threshold r_t. However, reputation is not the only means to categorize peers. Another parameter is *reciprocity*, defined as follows:

$$\sigma \doteq \frac{1}{1 + e^{-(S_p - S_r)}} \tag{5.3}$$

where S_p and S_r characterize a neighbor in terms of the total amount of provided and requested storage space, respectively. Like reputation, reciprocity is in $[0,1]$ and dimensionless. Peers can then decide the order by which serving incoming requests by ranking known peers according to reciprocity.

The Cloud is involved when available peers have low reputation or do not guarantee a minimum specific performance, and when the number of stored data chunks falls below a threshold n_t (such that $k < n_t < n$), because of neighbors departures, and no highly reputed peers are available. Every peer should be able to set its own r_t and n_t thresholds, according to its specific goals.

The collaborative distributed storage application enacts four functional policies.

The storeDataChunks policy is executed when a peer wants to store a new file, or to refresh an old one, by means of newly generated data chunks. The peer defines a list of suitable neighbors (according to one of the optimization strategies discussed later in this section) and tries to send n_i chunks to each of them. Also, the peer updates Table2 accordingly, once a chunk has been sent. After asking all highly reputed neighbors, remaining chunks are stored into the Cloud.

The retrieveStoredDataChunks policy is executed when a whole file has to be retrieved. The peer retrieves the list of neighbors storing data chunks of the file of interest from its Table2, and requests them. If the number of retrieved data chunks is less than k, the Cloud is asked to provide missing chunks.

The system provides two alternative approaches to refresh a stored file, one being *reactive*, the other being *proactive* [70]. Reactive maintenance requires the periodic execution of the monitorStoredFile policy , which may trigger a refreshStoredFile. The peer retrieves the list of neighbors storing chunks of the file of interest from its Table2, and requests them the chunks indexes. Such are compared to the ones in Table2 and, if different, the table is updated accordingly. After interrogating all neighbors, the peer checks if the total number of data chunks is lower than n_t and, if so, it generates new data chunks and executes the refreshStoredFile policy. Conversely, proactive maintenance consists in executing the refreshStoredFile policy after each retrieveStoredFile. If d data chunks are available (with $k \leq d < n_t < n$), the refreshStoredFile policy generates $n - d$ new data chunks which are stored by means of the storeNewFile policy. A file is unavailable if less than k data chunks can be retrieved. Unavailability is temporary when unavailable data chunks are hosted by peers that are currently offline, but will come back later. Otherwise, if unavailable data chunks have been deleted by their hosts, or if such hosts have left the network forever, data chunks are definitively lost.

The collaborative distributed storage application enacts two autonomic policies.

The updateReputation policy is described by Algorithm 1 and is executed after each store, retrieve or refresh interaction with a peer. Table1 stores r_1, r_2 and r. The updated reputation r is computed as a linear combination of r_1 and r_2, based on two application-specific weights.

The updateReciprocity policy is described by Algorithm 2 and is executed when a peer requests, or is requested, for storage space. Thus, the policy allows to keep trace in Table1 of the total amount of provided storage space S_p, and of the total amount of requested storage space S_r, for every known peer. Such values are used to compute the reciprocity, by means of eq. (5.3).

Every peer tries to minimize the economical cost of storing files in the system,

Algorithm 1 updateReputation

```
1: $n_r$ = table1.getNumRequestsTo(peer);
2: $n_{fr}$ = table1.getNumFulfilledRequestsBy(peer);
3: $rt_p$ = table1.getExpectedResponseTime(peer);
4: $rt_c$ = table1.getExpectedResponseTime(cloud);
5: $r_1 = n_{fr}/n_r$;
6: $r_2 = \min\{1, rt_c/rt_p\}$;
7: $w_1$ = this.getRequestWeight();
8: $w_2$ = this.getResponseTimeWeight();
9: $r$ = ($w_1$ * $r_1$ + $w_2$ * $r_2$) / 2;
10: table1.updateReputation(peer, $r$);
```

Algorithm 2 updateReciprocity

```
1: $S_p$ = table1.getProvidedSpace(peer);
2: $S_r$ = table1.getRequestedSpace(peer);
3: $\sigma$ = 1 / (1 + pow(e, ($S_r$ - $S_p$)));
4: table1.updateReciprocity(peer, $\sigma$);
```

subject to two constraints. First, the file availability must be at least A_{min}.[1] Second, the response time for file retrieval RT must not exceed RT_{max} given threshold. To meet such objectives, every peer relies on autonomic policies to sporadically update reputations and periodically optimize thresholds. If a centralized entity had, at any time, a comprehensive knowledge of the whole system, the following global optimization problem should be solved:

$$\begin{aligned}
&\underset{\mathbf{n}_c}{\text{minimize}} && \mathbf{C} = \mathbf{n}_c u_c \\
&\text{subject to} && \max\{rt_c \mathbf{n}_c, \max_{j=0,..,m_i} \{\mathbf{rt}_j^T \mathbf{n}_j\}\} \leq \mathbf{RT}_{max} \\
& && A_c(\mathbf{n}_c') + \mathbf{A}_p^T(\mathbf{1} - \mathbf{n}_c') \geq \mathbf{A}_{min}
\end{aligned}$$

where $\mathbf{n}_c$ is a vector whose i-th element represents the number of data chunks the i-th peer has to store in the Cloud, for a given file ($0 \leq n_c \leq n$); u_c is the economic cost required to store one data chunk in the Cloud; rt_c is the average response time the Cloud takes to store a data chunk; $\max_j\{\mathbf{rt}_j^T \mathbf{n}_j\}$ is a vector whose i-th element represents the average response time the i-th peer takes to store data chunks into

[1]A typical value for A_{min} is 99.99% (https://aws.amazon.com/s3/details/).

its neighbors (assuming they are heterogeneous); A_c is the availability of the Cloud (ideally we can assume $A_c = 1$); $\mathbf{A}_p$ is a matrix whose i-th diagonal element represents the average availability of the i-th peer's neighbors; $\mathbf{1}$ is a vector of 1s; $\mathbf{n}$ is a vector whose i-th element represents the number of data chunks the i-th peer has to publish; $\mathbf{n}'_c$ is a vector whose i-th element is the ratio between n_c and n of the i-th peer. The two constraints are independent.

Each peer has to solve the following local version of the problem:

$$\begin{aligned} \underset{n_c}{\text{minimize}} \quad & C = n_c u_c \\ \text{subject to} \quad & \max\{rt_c n_c, \max_{j=0,..,m}\{rt_j n_j\}\} \leq RT_{max} \\ & n_c \geq (nA_{min} - A_p)/(A_c - A_p) \end{aligned}$$

If peers are only interested in not exceeding a given C_{max} *budget*, a possible *adaptive* heuristic strategy consists of using constraints on C and rt to compute and update the minimum number of high-reputation peers m_t over which no data chunks are sent to the Cloud. If the measured file availability A is below A_{min}, the computed m_t is adjusted by adding $k(A_{min} - A)$, where k has to be set to find a tradeoff between increasing settling time and decreasing overshoot. If the actual number of high-reputation neighbors m is less than m_t, the peer computes the fraction of data chunks that should be stored into the Cloud as

$$\mu_c = \frac{n_c}{n} = 1 - \frac{m}{m_t}$$

Suppose there are m high-reputation peers (*i.e.*, peers whose reputation is $r \geq r_t$). r_t is interpreted as the ideal availability of the considered peer (*i.e.*, the minimum availability required to all peers to guarantee the desired file availability [70]). Table 5.3 reports the most interesting values.

At startup, new peers should be provided with a set of contacts of peers they can trust and should try to use them as storage providers. Also, the peer may ask them for contacts of highly reputed peers. By comparing received lists, the peer should be able to find a set of candidates to be used as storage providers.

File availability	Peer availability
0,99	0,64
0,999	0,69
0,9999	0,72
0,99999	0,75

Table 5.3: File availability values of major interest, with corresponding values for minimum required peer availability.

With reference to the adaptive strategy, the probability δ_c that less than m_t highly reputed known peers are online (*i.e.*, δ_c is the probability to use the Cloud) can be expressed as follows [48].

$$\delta_c \leq \psi \left[1 - \sum_{j=m_t+1}^{+\infty} f_D(j) \left(e^{-(m_t-1)H\left(\frac{m_t-1}{j},\phi\right)} - \phi^j \right) \right] \doteq \delta_c^* \qquad (5.4)$$

where D random variable denotes the number of known online peers (from the point of view of a generic peer), $\phi = P\{R > r_t\} = 1 - F_R(r_t)$ is the probability that the reputation is higher than r_t threshold (R is a continuous random variable representing the peers' reputation), and $\psi = P\{D > m_t\}$.

To evaluate the proposed NAM-based approach, we simulated the distributed storage system and compared the case of adaptive thresholds with those in which thresholds are fixed. To this purpose, we have used the DEUS general-purpose discrete event simulation environment.

The simulated cloud system is assumed to be always available and is characterized by prices per PUT/GET operations, respectively denoted as c_{PUT} and c_{GET}, and by an exponentially distributed response time with mean value rt_c. Such assumptions are in line with related work on P2P/Cloud systems [66, 71]. We have deployed in our Department a cloud prototype based on OpenStack and augmented with a dispatcher, a monitor and an autonomic auto-scaling module. The latter has the purpose of adapting the number of active virtual machines to the current workload.

Together with the simulated cloud, there are N nodes connected in a P2P network. We considered two models for its topology. The first one is based on non-preferential growth (*i.e.*, every new node connects to α randomly selected nodes), whose resulting node degree distribution is $P(k) = e(1-e^{-1/\alpha})e^{-k/\alpha}$, $\forall k > \alpha$ (exponential), and the mean node degree is $\langle k \rangle = e^{(1-1/\alpha)}/(1-e^{-1/\alpha})$. The second topology model is the one proposed by Barabási and Albert (*BA model*) [72], which is based on growth and preferential attachment to construct scale-free networks, and whose node degree distribution is $P(k) \simeq 2\alpha^2 k^{-3}$, $\forall k \geq \alpha$, and the mean node degree is $\langle k \rangle = 2\alpha$.

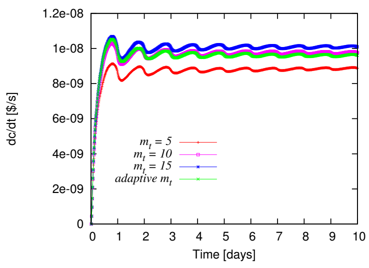
Every simulated day, all N peers stay online for a random time period, and only the ρ_3 fraction stays online for more than 3 hours. Sporadically (with inter-arrival time following an exponential distribution with mean value), a randomly selected peer stores a new file in the system by enacting the previously illustrated policies. Every stored file is periodically maintained (with period T). File retrieving is simulated with the same storing inter-arrival statistics. The reputation r of peers changes with time, based on the previously described rules.

Param	**Description**	**Value**
N	Total number of peers	1000
α	Peers' initial node degree	$\{5, 10, 15\}$
A_{min}	Threshold to consider file availability as high	0.9999
r_t	Threshold to consider a reputation value as high	0.72
m_t	Threshold for the number of high reputation neighbors	$\{5, 10, 15, adaptive\}$
k	Number of symbols of an information message	30
n	Number of symbols of an encoded information message	60
n_t	Threshold for the number of stored data fragments	45
β	Data chunk size	10^5 [Byte]
T	Maintenance period	$\{100, 300, 600, 1800\}$ [s]
rt_p	Average peer response time for 1 data chunk	1 [s]
rt_c	Average cloud response time for 1 data chunk	3 [s]
c_{PUT}	Cost per PUT operation in the Cloud	\$0.00001
c_{GET}	Cost per GET operation in the Cloud	\$0.000001
ρ_3	Fraction of peers that every day stay online at least 3 hours	$\{0.2, 0.5, 0.8\}$
w_1	Weight given to satisfied requests for reputation update	random $\in [1.2, 2]$
w_2	Weight given to response time for reputation update	random $\in [0, 1.2]$
λ	Publication rate	$\{5, 10\}[s^{-1}]$

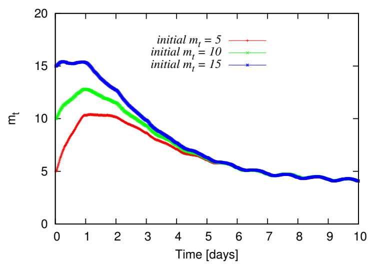
Table 5.4: Simulation parameters.

Table 5.4 lists all simulator's parameters. By using such values, the numerical solution of equation (5.4), which represents the cloud use probability upper bound, is $\delta_c^* = 0.64$, in case of non-preferential topology, and $\delta_c^* = 0.55$, in case of preferential topology.

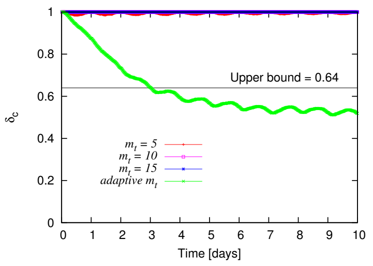
The performance of the adaptive strategy, with m_t evolving over time, has been compared to the case where a fixed m_t is used. For the former, m_t is initialized to 10. For the latter, instead, $m_t \in \{5, 10, 15\}$. Fig. 5.14 and 5.15 [48] present the evolution over time of dC/dt (average cost variation for each peer), of m_t with three different initialization values (only for the adaptive strategy), of the cloud use probability δ_c and of the average peer reputation r, for the non-preferential and preferential topologies, respectively. The considered scenario is characterized by $\lambda = 10$, $T = 1800$ [s], $\rho_3 = 0.2$ and $\alpha = 10$. The measured availability A always resulted to be 1 and is therefore not plotted. With the adaptive strategy, the cloud use probability δ_c tends to a value that is much lower than 1, thus respecting the upper bound provided by equation (5.4). Also, m_t converges to the same value, regardless of its initialization value.



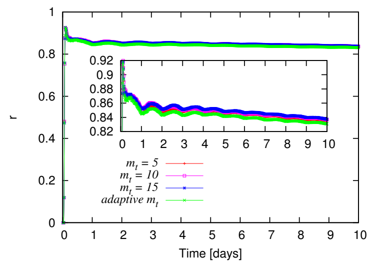
(a) Average cost variation for each peer



(b) Adaptive threshold m_t

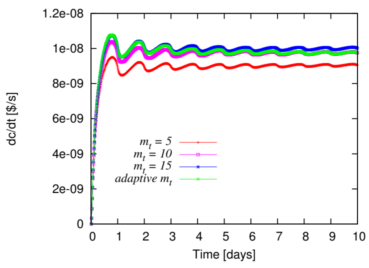


(c) Cloud use probability δ_c

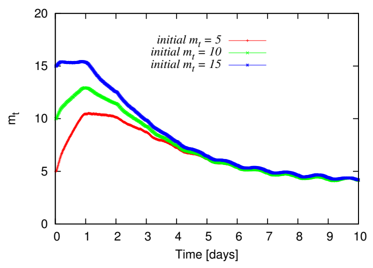


(d) Average peer reputation r

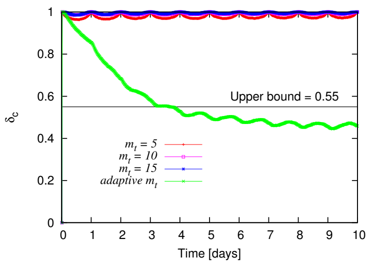
Figure 5.14: Evolution over time of the most interesting performance indicators and m_t, for the exponential topology. The measured availability is $A = 1$.



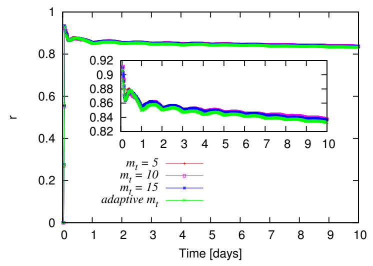
(a) Average cost variation for each peer



(b) Adaptive threshold m_t



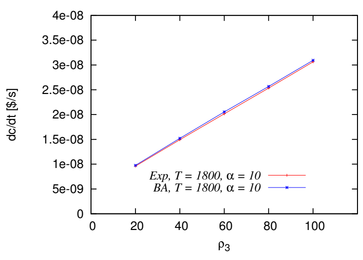
(c) Cloud use probability δ_c



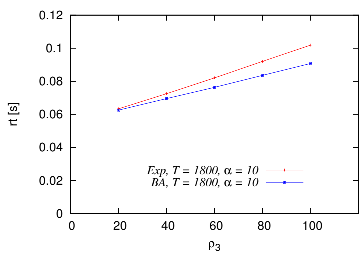
(d) Average peer reputation r

Figure 5.15: Evolution over time of the most interesting performance indicators and m_t, for the BA topology. The measured availability is $A = 1$.

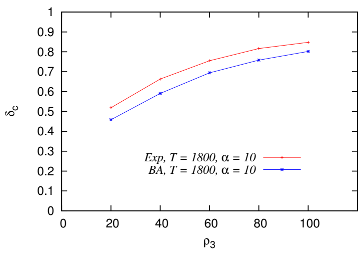
Fig. 5.16 [48] presents the evolution of performance indicators and m_t for different ρ_3 values. It can be noted that all increase with ρ_3. The reason is that, if the fraction ρ_3 of peers that every day stay online more than 3 hours increases, the overall stability of the P2P network increases as well, and is then reasonable to be more selective with neighbors. Therefore, the autonomic approach makes the threshold for the number of high reputation neighbors grow. The possibility to use better peers improves performance indicators. Interestingly, with the preferential topology, the cloud use probability δ_c and the response time are lower than with the non-preferential one. The reason is that the preferential topology is characterized by few highly connected peers, which therefore have a lot of options for storing data, while most peers have a limited amount of neighbors.



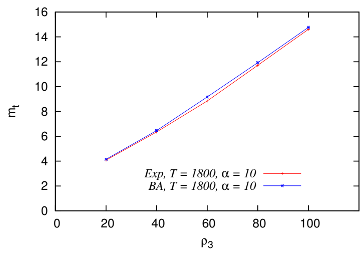
(a) Average cost variation for each peer



(b) Average peer response time *rt*



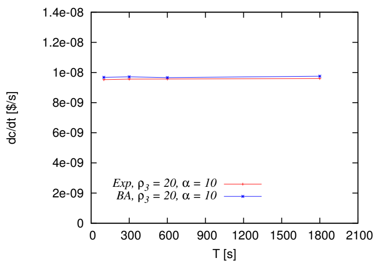
(c) δ_c



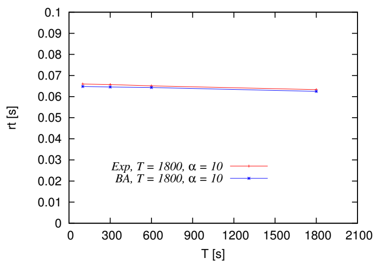
(d) Adaptive threshold m_t

Figure 5.16: Evolution of performance indicators versus ρ_3. The measured availability is $A = 1$.

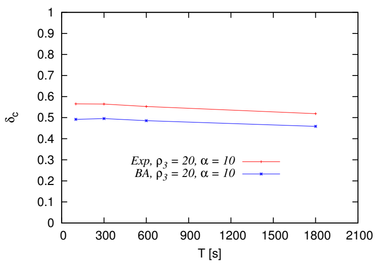
Fig. 5.17 [48] presents the evolution of performance indicators and m_t for different values of the maintenance period T. Values less than 1800 (the one used in the performance analysis) do not provide any particular advantage.



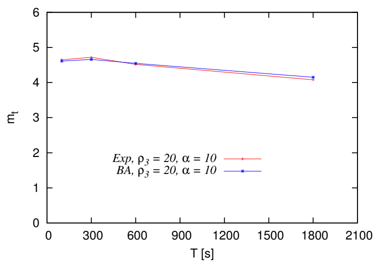
(a) Average cost variation for each peer



(b) Average peer response time *rt*



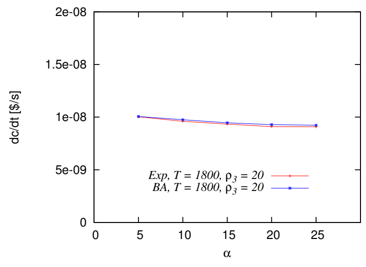
(c) δ_c



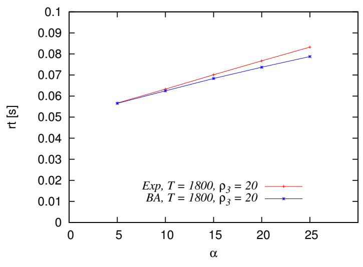
(d) Adaptive threshold m_t

Figure 5.17: Evolution of performance indicators versus T. The measured availability is $A = 1$.

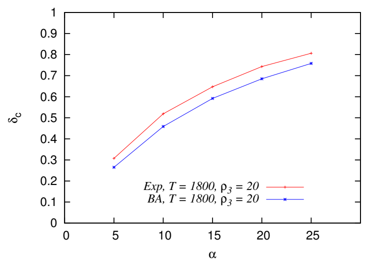
Finally, Fig. 5.18 [48] presents the evolution of performance indicators and m_t for different values of the initial node degree of each peer α. It can be noted that m_t and all performance indicators increase with α. The reason is the same that justifies the behavior of the system facing the ρ_3 increase, in that more available neighbors allows peers to be more selective. As a consequence, the average performance of the system increases.



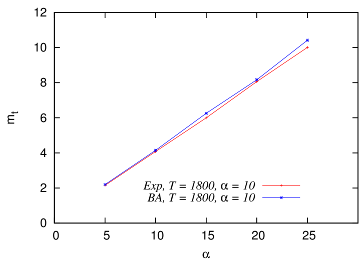
(a) Average cost variation for each peer



(b) Average peer response time rt



(c) δ_c



(d) Adaptive threshold m_t

Figure 5.18: Evolution of performance indicators versus α. The measured availability is $A = 1$.

5.3 Urban participatory sensing

Participatory Sensing is the process whereby context data and information are acquired by users and shared or analyzed within the community to produce knowledge and services. Participatory Sensing applications benefit from the wide diffusion of mobile phones which are not only communication means and powerful multimedia devices, as they also include a large variety of sensors such as GPS receiver, microphone, temperature, light detector and accelerometer. The application of Participatory Sensing to urban areas, namely Urban Participatory Sensing (UPS), provides advantages over traditional sensor networks. Primarily, since users are usually (and irregularly) distributed over a city, UPS applications can cover most parts of the urban space with a minimum expense due to the fact they leverage the presence of existing mobile devices (smartphones and tablets) and communication infrastructures (cellular or Wi-Fi).

Typical Participatory Sensing applications present a centralized architecture in which volunteer users collect data and send it to a central server for processing. Data collection can be autonomous or triggered by the user, and based on either context or temporization. The central server usually analyzes and processes received data and provides useful services or information to interested users and systems.

In a decentralized UPS architecture instead, sensed data is made available to everyone and no central server is strictly required, although a remote cloud can be used to perform computationally-intense processing. Fig. 5.19 presents the actors and functions in the proposed autonomic approach, where network nodes can play the role of information consumers and/or providers. Collected data get aggregated and processed by users or by a remote cloud. The resulting information is then presented to interested peers. For example, public health applications, aiming at analyzing the effects of the environment on health issues, can take advantage of a remote cloud to aggregate remarkable amounts of data and processing. On the other hand, there exist classes of applications, such as public transportation monitoring, which require less aggregation and processing and therefore do not need the computational performance provided by a remote cloud. A publish/subscribe mechanism can be used to let users show interest in events, such as the current location of a bus and delay estimate while currently traveling users provide related information.

Since users continuously join and leave the network, autonomic self-management policies, handled by NAM4J rules engine, allow the developed application to detect if the number of providers is below an application-specific threshold and react to preserve the quality of the service by performing recruitments among active peers.

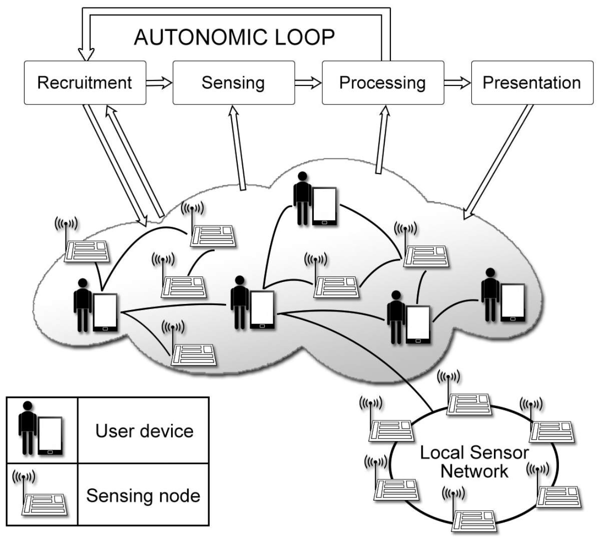


Figure 5.19: Architecture of the proposed autonomic approach to UPS.

This happens by checking if a certain device satisfies minimum requirements to be a provider (*i.e.* available sensors and computational capabilities specified in the JSON item descriptor presented in Section 4.2) and the user agrees to assume such a role. If requirements are met, but the device lacks necessary code, a peer sends it using NAM4J mobility features. In fact, the copy mobility action, which allows the migration of functional modules and services between peers, can be used to autonomically modify the behavior of nodes at runtime. It is also used to send to new users the functional module which implements networking management, thus allowing the node to join the network.

The developed application allows users to retrieve and publish different types of sensed information. Fig. 5.20 presents the structure of the nodes.

Each node in the network is provided with the following functional modules.

- A network manager functional module (f_N) - it allows the NAM to participate in a P2P network, with publish and lookup services (s_p and s_l, respectively). We have developed functional modules which allow the deployment of Chord,

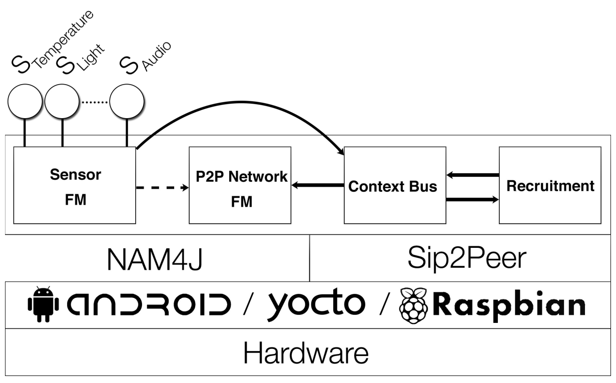


Figure 5.20: Layered structure of network nodes.

Kademlia, full mesh and random graph topologies;

- SensorFunctionalModule (f_S) - it uses the publish service of f_N to share context events related to sensed information;
- ReasonerFunctionalModule (f_R) - it uses the lookup service of f_N to get context events of interest. To allow f_N to behave in a proactive way, f_R exposes a notify service (s_n) that is called by f_N when relevant context events are produced either by f_S or by remote NAMs.

Among such functional modules, f_R is the most demanding in terms of CPU cycles, f_N is mostly bandwidth-consuming and f_S is a thin software layer that consumes the battery of the mobile device as long as it uses local sensors. To reduce the energy consumption, the best candidate for being offloaded or migrated is f_R, as f_N must stay on the mobile device for connectivity reasons, and f_S can reduce its operation rate. Instead, f_R is always active to provide aggregated sensing information to the user, but it is not mandatory for it to run on the mobile device. It is worth noting that copying f_N to other NAMs allows them to join the overlay network thus increasing its size, which may be useful to balance the communication and information storage workload.

We can consider an augmented execution scenario in which a mobile device hosting a NAM runs short of a certain resource (*e.g.*, battery power or CPU cycles), while f_R is performing its demanding task. In general, an autonomic MCC application should react accordingly to the situation, so that a task is completed successfully, even when local resources become insufficient, and without requiring user intervention. In our scenario, possible decisions are to offload f_R for execution on the cloud or on another mobile node.

It is crucial to identify effects of such decisions, mechanisms to enact them, as well as the responsibility of related mobility operations (*e.g.*, the back action). Regarding the offloading decision, a reasonable solution is to rely on NAM self-management policies. These are designated of monitoring events related to the device state, in order to preserve a given quality of service or safety conditions. In the following, the role of f_R policies for our specific example is analyzed.

The f_R policies are the following ones.

$$\begin{aligned}\mathscr{P}_o = \{&(\mathsf{cpuLoadUpdate}, load > 70\%, \mathsf{offload}(\mathit{fid})),\\ &(\mathsf{batteryChargeUpdate}, charge \leq 30\%, \mathsf{offload}(\mathit{fid}))\}\end{aligned}$$

$$\mathscr{P}_l = \{(\mathsf{wifiConnectionReport}, quality < 4, \mathsf{back}(\mathit{fid}))\}$$

$$\mathscr{P}_r = \{(\mathsf{serviceQualityReport}, quality < 7, \mathsf{go}(\mathit{fid}))\}$$

fid is the identifier of f_R. On-site policies $\mathscr{P}_o$ monitor the availability of CPU and battery resources and, if necessary, trigger the offloading action to reduce resource consumption. When the offloading completes, the policy handler is split into a local and a remote handler (executing, respectively, $\mathscr{P}_l$ and $\mathscr{P}_r$). The former monitors the quality of the wireless connection and decides when it is necessary to request the module back from the cloud service because the connection has become unreliable while not loosing the computation performed so far. The latter resides on the cloud service NAM and monitors the quality of the computation. If such is not satisfactory, offloading may become a disadvantage and the module may decide to go to another NAM, possibly the original one.

The behavior of the remote cloud service, which provides elastic resources to registered users having a positive credit balance, is analyzed in the following. Such a service provides the users with one or more virtual machines running a cloned system image. The mobile device can offload f_R to a cloned replica for remote execution, thus saving battery and time as the cloud speeds up the computation. The offloading process starts on the mobile device when enacted by f_R policies. NAMs hosted on virtual machines are instead characterized by policies which perform other monitoring tasks, such as the ones described by the following rules.

$$\begin{aligned}\mathscr{P} = \{&(\mathsf{cpuLoadUpdate}, load > 80\%, LoadBalance),\\ &(\mathsf{accountCreditReport\_}\mathit{fid}, credit = 0, \mathsf{go}(\mathit{fid}))\}\end{aligned}$$

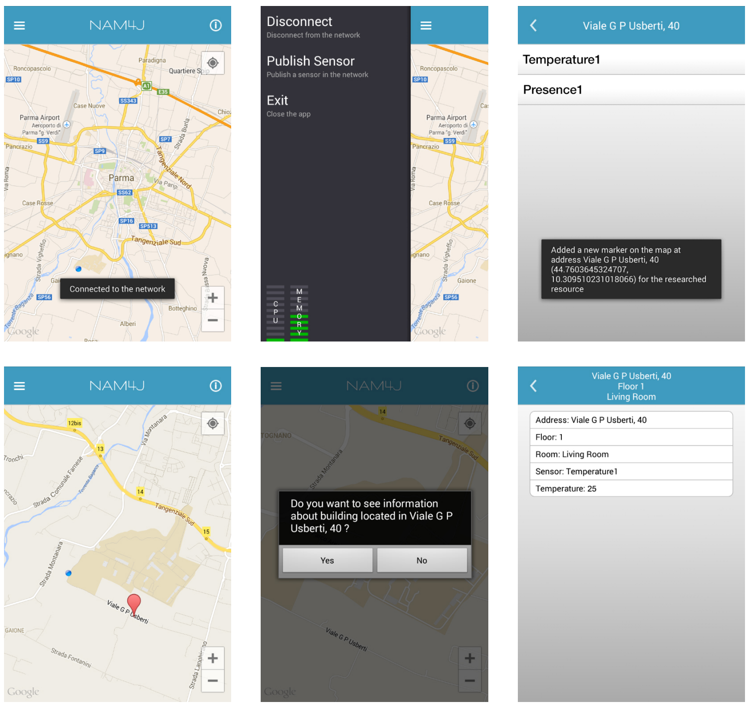
The CPU load is monitored so that, when it gets too high, a balancing action is executed by moving a functional module to another virtual machine. Furthermore, the user credit is monitored so that, when it gets insufficient, hosted functional modules are sent back to the owner.

Regarding on-site f_N policies, the following ones regulate the module replication on other NAMs.

$$\begin{aligned}&(\mathsf{networkWorkloadUpdate}, workload > 70\%, \mathsf{copy}(\mathit{fid}')),\\ &(\mathsf{networkReq}, true, \mathsf{copy}(\mathit{fid}'))\end{aligned}$$

fid' is the identifier of f_N. The first policy monitors the workload of the networking module and, when it becomes overloaded, executes a copy action to activate a new network node on another NAM. The second policy performs the same action when a copy of the networking module is explicitly requested by another NAM that desires to be part of the network.

To enable the interaction between NAM and sensor layers, f_S has been designed to aggregate services with the purpose of providing values collected by physical sensors. Leveraging on NAM modular approach, we implemented a specific service for each sensor type. For our experimental setup we have deployed NAM nodes on mobile and desktop devices, Raspberry Pi and Intel Galileo boards. We have also developed an Android app, presented by Fig. 5.21 [41], to allow users to publish and retrieve sensed data. The architecture provides code reuse since the functional module employed on all nodes is the same. However, services are different, in that related code is platform-dependent.



Connected to the network

Added a new marker on the map at address Viale G P Usberti, 40 (44.7603645324707, 10.309510231018066) for the researched resource

Do you want to see information about building located in Viale G P Usberti, 40 ?

Viale G P Usberti, 40
Floor 1
Living Room

Address: Viale G P Usberti, 40
Floor: 1
Room: Living Room
Sensor: Temperature1
Temperature: 25

Figure 5.21: Screenshots of the Android application. The user is allowed to publish information acquired by her/his device. Also, the user can search for deployed sensors and be notified of updates.

As illustrated by Fig. 5.20, f_S uses the context bus, described in section 4.3, to publish information about events, and the context bus uses f_N to send related messages to other peers. Also, each NAM manages the reputation of known peers using DARTSense, presented in section 4.4.

The autonomic policy used to recruit new information providers, which is illustrated by Algorithm 3, strongly relies on the publish/subscribe mechanism enabled by the distributed context bus. When a node is interested in receiving event notifications, it saves the current time, then subscribes to the upsEvent (*e.g.*, CO_2 level updates) in a specific area (represented by the georefArea parameter) and starts a timer. The onTimeUp function is executed when the specified time interval MAX_TIME has passed and no notification for the upsEvent has been received. The lastEventReceivedTime variable represents the timestamp of the last received event notification. The onTimeUp function checks if the number of known providers for the area of interest is below a specified threshold and, if it is, it subscribes the node to the availabilityEvent. Such event notifies the availability of peers willing to become information providers in a specified area. When an event notification is received, function onReceivedEvent is executed. First, it checks whether the node subscribed to the related event and, if so, it checks if the event type is availabilityEvent. If this is the case, the node checks whether the available peer has enough resources to run required sensing functional modules and services. If it is, the node sends such items to the peer and updates the number of known providers for the specified area. If the received notification is instead related to another type of event (*e.g.*, sensed data), the node properly handles it. In either case, the timer is reset. Finally, the node checks if other peers subscribed to the event and notifies them.

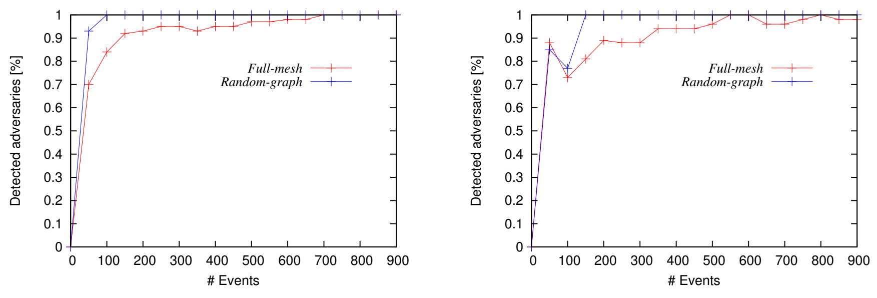
We have performed several experiments on the PlanetLab[2] platform, to test and evaluate the DARTSense reputation update algorithm. We have deployed 100 peers, 90 of them being set as *participants* (*i.e.*, peers that always provide right information), and 10 as adversaries (*i.e.*, peers that provide right information with probability p and wrong information with probability $1-p$). The purpose of the experiments was to check whether and in how much time peers detect malicious participants and assign them a low reputation, for different values of p and network topology (full mesh and random graph). We have performed tests for different p values, being $p = 0.2$ and $p = 0.8$ the most interesting cases. When a full mesh topology is used, we observed that the earliest generated events have more influence on the time required to classify peers as good or bad. Fig. 5.22 (a) and 5.22 (b) show that the percentage of detected adversaries increases with the total number of produced events. Apparently, the random graph topology performs better. The reason is the fact that, in full mesh

[2]https://www.planet-lab.org/

Algorithm 3 Autonomic recruitment policy

```
 1: while interestedInReceivingNotifications(upsEvent) do
 2:     lastEventReceivedTime ← getCurrentTime();
 3:     subscribe(upsEvent, georefArea);
 4:     setTimer(lastEventReceivedTime + MAX_TIME, onTimeUp(upsEvent, georefArea));
 5: function ONTIMEUP(upsEvent, georefArea)
 6:     if getProvidersNumber(georefArea) ≤ THRESHOLD then
 7:         subscribe(upsAvailabilityEvent, georefArea);
 8: function ONRECEIVEDEVENT(event)
 9:     if eventSubscribed(event) then
10:         if event == 'upsAvailabilityEvent' then
11:             if checkSpecs(event.publisher, requirements) then
12:                 sendFMAndServicesTo(event.publisher);
13:                 updatesProvidersNumber(georefArea);
14:         lastEventReceivedTime ← getCurrentTime();
15:         setTimer(lastEventReceivedTime + MAX_TIME, onTimeUp(upsEvent, georefArea));
16:         // Event management
17:     notifySubscribers(event);
```

networks, all adversaries interact with all nodes while, in random graphs, not all the adversaries receive a subscribe message. Therefore, peers in a full mesh try to detect all adversaries, while peers in a random graph only a subset (*i.e.*, 20% on the average).



(a) $p = 0.2$ (b) $p = 0.8$

Figure 5.22: Percentage of detected adversaries versus the total number of published events.

Chapter 6

Conclusions

In this thesis, we have presented the formal definition of a novel Mobile Cloud Computing (MCC) framework based on Networked Autonomic Machine (NAM), a general-purpose conceptual tool which describes large-scale distributed autonomic systems and is suitable for MCC applications as well, as it supports code, data and execution state mobility concepts. The definition of the framework aims at addressing the major issues and challenges that hinder the full realization of MCC systems, namely: the lack of an agreed upon conceptual model for MCC systems; the fact that most of current applications are statically partitioned; the possibility of rapid changes in network conditions and local resource availability; privacy and security concerns. The introduction of autonomic policies in the MCC paradigm proved to be a promising technique to increase the robustness and flexibility of MCC systems. In particular, autonomic policies based on continuous resource and connectivity monitoring help automate context-aware decisions for computation offloading.

Before the PhD activity, a Java implementation of the NAM framework (NAM4J) existed, and only included a reduced set of core classes which allowed to implement NAM entities with limited capabilities. During the activity we have provided NAM with a formalization in terms of a *transformational operational semantics* in order to fill the gap between its implementation and its conceptual definition. The activity has been carried out in collaboration with a research team of the *IMT Institute* *for Advanced Studies Lucca*. The formalization process clarified several aspects and allowed us to refine the NAM framework with specific focus on MCC features. In general, NAM provides functionalities as pluggable components, defined as functional modules and services, whose code, execution state and data can be migrated among network nodes.

Moreover, we have extended and enriched NAM4J by adding several components with the purpose of managing large scale autonomic distributed environments. In particular, the middleware allows for the implementation of peer-to-peer (P2P) networks of NAM nodes. Moreover, NAM mobility actions have been implemented to enable the migration of code, execution state and data.

Within NAM4J, we have designed and developed a component, denoted as context bus, which is particularly useful in collaborative applications in that, if replicated on each peer, it instantiates a virtual shared channel allowing nodes to notify and get notified about context events. Such events may represent changes that happen in the environment and may influence the behavior of the distributed system.

Regarding the autonomic policies management, we have provided NAM4J with a rule engine, based on JBoss Drools, whose purpose is to allow a system to autonomously determine when offloading is convenient.

We have also provided NAM4J with trust and reputation management mechanisms to make the middleware suitable for applications in which such aspects are of great interest. An example is represented by collaborative applications, such as participatory sensing, where privacy is one of the main concerns. Tagging sensed data with location and time can involuntarily reveal personal information. On the other hand, users need to trust received information, and a completely anonymous system can hinder data reliability. NAM4J's trust and reputation management mechanisms are based on the *Anonymous Reputation and Trust Sensing (ARTSense)* framework, available in the literature. While the ARTSense architecture is centralized (*i.e.*, a server hosts a database to store reputation levels) and, therefore, has well-known drawbacks, such as the presence of a single point of failure and scalability issues, we have designed and implemented a distributed version of the framework, denoted as DARTSense, where no central server is required, as reputation values are stored and updated by participants in a subjective fashion.

As part of the PhD activity, we have investigated the literature regarding MCC systems. The design of such systems is a challenging task, in that both the mobile device and the Cloud have to find energy-time tradeoffs, and the decisions taken by one part affect the other part. The analysis pointed out that all MCC models focus on mobile devices, and consider the Cloud as a system with unlimited resources. To contribute in filling this gap, we defined a modeling and simulation framework for the design and analysis of MCC systems, encompassing both their sides. We have also implemented a simulator of the model which allows the simulation of large scale MCC systems provided with autonomic policies. The simulator is highly modular and reusable in that it is possible to define different mobile device classes (each characterized by specific features), to configure the most appropriate characterization of the communication channel between the mobile device and the Cloud, to define any kind of policy the Cloud enacts to distribute tasks among its virtual machines (VMs), to define any kind of policy VMs enact to extract tasks from their queues, and to set an adaptive policy that updates the number of active VMs with the purpose of balancing performance and cost.

We have applied the NAM principles to two different application scenarios.

First, we have defined a hybrid P2P/cloud approach where components and protocols are autonomically configured according to specific target goals, such as cost-effectiveness, reliability and availability. The superior availability of the Cloud makes it a more appealing environment for firms when compared to the best effort philosophy of P2P. However, the Cloud alone may be not sufficient to achieve cost-effective and efficient large scale distributed collaborative environments. Cloud and P2P approaches are widely different in that, while the former is based on completely decentralized protocols, the latter follows the client/server paradigm to leverage on the presence of high performance data centers. Also, P2P services are usually free and do no guarantee reliability nor high performance, while cloud services are usually fee-based and performance is contracted. Merging the two paradigms brings together the advantages of both: high availability, provided by the Cloud presence, and low cost, by exploiting inexpensive peers resources. As an example, we have shown how the proposed approach can be used to design NAM-based collaborative storage systems based on an autonomic policy to decide how to distribute data chunks among peers and Cloud, according to cost minimization and data availability goals.

As a second application, we have defined an autonomic architecture for decentralized urban participatory sensing (UPS) which bridges sensor networks and mobile systems to improve effectiveness and efficiency. The developed application allows users to retrieve and publish different types of sensed information by using the features provided by NAM4J's context bus. Trust and reputation is managed through the application of DARTSense mechanisms. Also, the application includes an autonomic policy that detects areas characterized by few contributors, and tries to recruit new providers by migrating code necessary to sensing, through NAM mobility actions. The design of the application has been guided by the need to avoid that a central node could potentially become a bottleneck or a single point of failure for the whole system, because of its responsibility on a large subset of the network. Also, we wanted to reduce the risk of misleading information dissemination, while preserving users' privacy.

6.1 Further work

Several research opportunities emerge from the work presented in this thesis. This section introduces the most promising directions.

The first direction concerns the extension of the language for expressing autonomic policies. Although the Drools rule engine is a well-known and largely adopted approach, it has some limitations, mainly concerning policy composition. Thus, we could consider languages designed to express more structured and sophisticated forms of policies, such as XACML and FACPL.

The second direction regards the exploration of alternative solutions to the execution state migration problem. NAM4J is based on Java serialization, whose implementation is straightforward. However, as described in Chapter 4, serialization presents a number of drawbacks which hinder its application to the full range of possible MCC scenarios.

The third direction has the purpose of extending the proposed MCC model by further developing the concept of adaptive loop presented in Chapter 5. In fact, feedback interactions between mobile devices and the Cloud can be further improved, with the purpose to enforce adaptive behavior on both sides. In particular, as the statistics of the workload may change over time, Cloud policies enabling the adaptation of the number of VMs are important in order to satisfy the service level agreements (SLAs) with the clients, while reducing costs. We plan to investigate auto-scaling approaches based on control theory and global optimization strategies. Also, we plan to study the impact of communicating VMs and three-tiers applications where cloudlets are placed between mobile devices and remote clouds.

Appendix A

KLAIM Semantics

This appendix summarizes the key features of KLAIM formal language, which has been specifically designed to provide programmers with primitives to handle physical distribution, scoping and mobility of processes. Although KLAIM is based on process algebras, it makes use of *Linda*-like asynchronous communication and models distribution via multiple shared tuple spaces. Linda [73] is a coordination paradigm rather than a language, as it only provides a set of coordination primitives. It relies on the so-called generative communication paradigm, which decouples communicating processes both in space and time. Communication is achieved by sharing a common tuple space, where processes *insert*, *read* and *withdraw* tuples. The data retrieval mechanism uses pattern-matching to find the required data in the tuple space.

KLAIM, whose syntax[1] is presented by Table A.1, enriches Linda primitives with explicit information about the location where processes and tuples are allocated.

Nets N are finite collections of nodes composed by means of the parallel operator $N_1 \parallel N_2$. It is possible to restrict the scope of a name s by using the operator $(\nu s)N$. In the $N_1 \parallel (\nu s)N_2$ net, the effect of such an operator is to make s invisible within N_1.

Nodes $s ::_\rho C$ have a unique *location name* s (*i.e.*, their network address) and an

[1]During the NAM formal definition process, we used a version of KLAIM enriched with high-level features, such as assignments, standard control flow constructs and non-blocking retrieval actions, that simplify the modeling task. Such constructs are directly supported by KLAIM related tools such as, *e.g.*, the *Stochastic Analyser for Mobility (SAM)* tool (http://rap.dsi.unifi.it/SAM/).

Table A.1: KLAIM syntax.

(Nets)

$N \ ::= \ s ::_\rho C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu s)N$

(Components)

$C \ ::= \ P \quad | \quad \langle t \rangle \quad | \quad C_1 | C_2$

(Processes)

$P \ ::= \ a \quad | \quad X \quad | \quad A(p_1, \ldots, p_n)$

$\quad | \quad P_1 ; P_2 \quad | \quad P_1 | P_2 \quad | \quad P_1 + P_2$

$\quad | \quad \textbf{if}\ (e)\ \textbf{then}\ \{P_1\}\ \textbf{else}\ \{P_2\}$

$\quad | \quad \textbf{while}\ (e)\ \{P\}$

(Actions)

$a \ ::= \ \textbf{in}(T)@\ell \quad | \quad \textbf{read}(T)@\ell \quad | \quad \textbf{out}(t)@\ell$

$\quad | \quad \textbf{inp}(T)@\ell \quad | \quad \textbf{readp}(T)@\ell \quad | \quad \textbf{eval}(P)@\ell$

$\quad | \quad \textbf{newloc}(s) \quad | \quad x := e$

(Tuples)

$t \ ::= \ e \quad | \quad \ell \quad | \quad P \quad | \quad t_1, t_2$

(Templates)

$T \ ::= e \quad | \quad \ell \quad | \quad ?x \quad | \quad ?l \quad | \quad ?X \quad | \quad T_1, T_2$

allocation environment ρ, and host a set of components C. The *allocation environment* provides a name resolution mechanism by mapping *location variables* l (*i.e.*, aliases for addresses), found in processes hosted by the corresponding node, into locations s. The distinguished location variable **self** is used by processes to refer to the address of their current hosting node. *Components* C are finite plain collections of processes P and evaluated tuples $\langle t \rangle$, which are composed by means of the parallel operator $C_1 \,|\, C_2$.

Processes P are the KLAIM active computational units, which can be concurrently executed either at the same location or at different locations. They are built up from basic actions a, process variables X, and process calls $A(p_1, \ldots, p_n)$, by means of sequential composition $P_1;P_2$, parallel composition $P_1 \,|\, P_2$, non-deterministic choice $P_1 + P_2$, conditional choice **if** (e) **then** $\{P_1\}$ **else** $\{P_2\}$, iteration **while** (e) $\{P\}$, and (possibly recursive) process definition $A(f_1, \ldots, f_m) \triangleq P$, where A denotes a process identifier, while f_i and p_j denote formal and actual parameters, respectively. Hereafter, we do not explicitly represent process definitions and assume that they are available at any location of a net. Notably, e ranges over *expressions*, which contain basic types (boolean, integer, string, float, etc.) and values x, and are formed by using standard operators on basic values, and the non-blocking retrieval actions **inp** and **readp** (described in the following). In the remainder of this section, we use the notation ℓ to range over location names s and location variables l.

During their execution, processes perform basic *actions*, such as $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$, which represent retrieval actions and allow to withdraw/read data tuples from the tuple space hosted at a (possibly remote) location ℓ. Such actions exploit *templates* as patterns to select tuples in shared tuple spaces. Templates are sequences of actual and formal fields, where the latter are denoted as $?x$, $?l$ or $?X$ and are used to bind variables to values, location names or processes, respectively. Actions $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ are non-blocking versions of the retrieval actions. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read**, and additionally return the *true* value. Otherwise, the actions return the *false* value, and the executing process does not block. $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is ignored). Action $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space of the target node identified by ℓ, while action $\mathbf{eval}(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Both **out** and **eval** are non-blocking actions. Finally, action **newloc** creates new network nodes, while action $x := e$ assigns the value of e to x. Differently from all other actions, these are not provided with an address since they always act locally.

In the following, an example providing further details concerning the communication between KLAIM nodes, is presented. Let us consider the following KLAIM net:

$$\begin{array}{l} s_1 ::_{\{\mathbf{self} \mapsto s_1\}} \mathbf{out}(\mathsf{foo}, 5)@s_2;\ P_1 \\ \parallel\ s_2 ::_{\{\mathbf{self} \mapsto s_2\}} \mathbf{in}(\mathsf{foo}, ?x)@\mathbf{self};\ P_2 \end{array}$$

Since the process on s_2 node is blocked (due to the blocking semantics of the retrieval action **in**), the only possible evolution of the net is as follows:

$$\begin{array}{l} s_1 ::_{\{\mathbf{self} \mapsto s_1\}} P_1 \\ \parallel\ s_2 ::_{\{\mathbf{self} \mapsto s_2\}} (\langle \mathsf{foo}, 5 \rangle \mid \mathbf{in}(\mathsf{foo}, ?x)@\mathbf{self};\ P_2) \end{array}$$

The **out** action is performed and, as a result, the $\langle \mathsf{foo}, 5 \rangle$ tuple is inserted in the tuple space of the target node s_2. The presence of such a tuple triggers the execution of the **in** action:

$$s_1 ::_{\{\mathbf{self} \mapsto s_1\}} P_1 \ \parallel\ s_2 ::_{\{\mathbf{self} \mapsto s_2\}} P_2[5/x]$$

In fact, the $(\mathsf{foo}, ?x)$ template, which is the argument of the **in** action, matches the $\langle \mathsf{foo}, 5 \rangle$ tuple, thus binding value 5 to the variable x in the continuation process P_2.

Bibliography

[1] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski. *Cloud computing: principles and paradigms*, volume 87. John Wiley & Sons, 2010.

[2] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya. Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges. *Communications Surveys Tutorials, IEEE*, 16(1):337–368, First 2014. `doi:10.1109/SURV.2013.070813.00285`.

[3] Mohsen Sharifi, Somayeh Kafaie, and Omid Kashefi. A survey and taxonomy of cyber foraging of mobile devices. *Communications Surveys Tutorials, IEEE*, 14(4):1232–1243, Fourth 2012. `doi:10.1109/SURV.2011.111411.00016`.

[4] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013. URL: `http://dx.doi.org/10.1002/wcm.1203`, `doi:10.1002/wcm.1203`.

[5] Mazliza Othman and Stephen Hailes. Power conservation strategy for mobile computers using load sharing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):44–51, January 1998. URL: `http://doi.acm.org/10.1145/584007.584011`, `doi:10.1145/584007.584011`.

[6] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Saving portable computer battery power through remote process execu-

tion. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):19–26, January 1998. URL: `http://doi.acm.org/10.1145/584007.584008`, `doi:10.1145/584007.584008`.

[7] Rajesh Krishna Balan, Mahadev Satyanarayanan, So Young Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 273–286, New York, NY, USA, 2003. ACM. URL: `http://doi.acm.org/10.1145/1066116.1066125`, `doi:10.1145/1066116.1066125`.

[8] Jason Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 61–66, May 2001. `doi:10.1109/HOTOS.2001.990062`.

[9] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, Aug 2001. `doi:10.1109/98.943998`.

[10] D. Kovachev and R. Klamma. Beyond the client-server architectures: A survey of mobile cloud techniques. In *Communications in China Workshops (ICCC), 2012 1st IEEE International Conference on*, pages 20–25, Aug 2012. `doi:10.1109/ICCCW.2012.6316468`.

[11] D. Da Silva. Opportunities for autonomic behavior in mobile cloud computing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, Keynote talk at ICACÕ13. Available at https://www.usenix.org/conference/icac13/title-tba-0.

[12] Michele Amoretti, Alessandro Grazioli, Valerio Senni, Francesco Tiezzi, and Francesco Zanichelli. A formalized framework for mobile cloud computing. *Service Oriented Computing and Applications*, 9(3-4):229–248, 2015.

URL: `http://dx.doi.org/10.1007/s11761-014-0169-3`, `doi:10.1007/s11761-014-0169-3`.

[13] Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. *Mastering Cloud Computing: Foundations and Applications Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. URL: `http://doi.acm.org/10.1145/1721654.1721672`, `doi:10.1145/1721654.1721672`.

[15] Mahadev Satyanarayanan, P. Bahl, R Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, Oct 2009. `doi:10.1109/MPRV.2009.82`.

[16] Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. *Peer-to-peer computing: Principles and applications*. Springer Science & Business Media, 2009.

[17] IBM Corporation. An architectural blueprint for autonomic computing. *White Paper*, June 2006.

[18] M. Amoretti, F. Zanichelli, and G. Conte. Qualitative and quantitative modeling of policy-based autonomic systems, 2011.

[19] Mohammad Reza Nami and Mohsen Sharifi. A survey of autonomic computing systems. In *Intelligent Information Processing III*, pages 101–110. Springer, 2007.

[20] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing&mdash;degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, August 2008. URL: `http://doi.acm.org/10.1145/1380584.1380585`, `doi:10.1145/1380584.1380585`.

[21] Brian Melcher and Bradley Mitchell. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technology Journal*, 8(4), 2004.

[22] Joseph P. Bigus, Don A. Schlosnagle, Jeff R. Pilgrim, W Nathaniel Mills III, and Yixin Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[23] Gail Kaiser, Janak Parekh, Philip Gross, and Giuseppe Valetto. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 22–30. IEEE, 2003.

[24] Janak Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[25] Ya-Yunn Su and Jason Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 79–92, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1067170.1067180`, `doi:10.1145/1067170.1067180`.

[26] M.D. Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 217–226, March 2010. `doi:10.1109/PERCOM.2010.5466972`.

[27] A.F. Murarasu and T. Magedanz. Mobile middleware solution for automatic reconfiguration of applications. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1049–1055, April 2009. `doi:10.1109/ITNG.2009.194`.

[28] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems,*

2002. Proceedings. 22nd International Conference on, pages 217–226, 2002. doi:10.1109/ICDCS.2002.1022259.

[29] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 272–285. ACM, 2007.

[30] Ermyas Abebe and Caspar Ryan. Adaptive application offloading using distributed abstract class graphs in mobile environments. *Journal of Systems and Software*, 85(12):2755–2769, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0164121212001653, doi:http://dx.doi.org/10.1016/j.jss.2012.05.091.

[31] Verdi March, Yan Gu, Erwin Leonardi, George Goh, Markus Kirchberg, and Bu Sung Lee. µcloud: Towards a new paradigm of rich mobile applications. *Procedia Computer Science*, 5:618 – 624, 2011. The 2nd International Conference on Ambient Systems, Networks and Technologies (ANT-2011) / The 8th International Conference on Mobile Web Information Systems (MobiWIS 2011). URL: http://www.sciencedirect.com/science/article/pii/S1877050911004054, doi:http://dx.doi.org/10.1016/j.procs.2011.07.080.

[32] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1814433.1814441, doi:10.1145/1814433.1814441.

[33] Cong Shi, Vasileios Lakafosis, Mostafa H. Ammar, and Ellen W. Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the Thirteenth ACM International Symposium*

on Mobile Ad Hoc Networking and Computing, MobiHoc '12, pages 145–154, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2248371.2248394`, `doi:10.1145/2248371.2248394`.

[34] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[35] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[36] Gonzalo Huerta-Canepa and Dongman Lee. An adaptable application offloading scheme based on application behavior. In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 387–392. IEEE, 2008.

[37] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013. URL: `http://dx.doi.org/10.1002/wcm.1203`, `doi:10.1002/wcm.1203`.

[38] M. Amoretti, M. Picone, and F. Zanichelli. Global ambient intelligence: An autonomic approach. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pages 842–847, March 2012. `doi:10.1109/PerComW.2012.6197629`.

[39] A. Grazioli, M. Picone, F. Zanichelli, and M. Amoretti. Code migration in mobile clouds with the nam4j middleware. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, volume 2, pages 194–199, June 2013. `doi:10.1109/MDM.2013.93`.

[40] Michele Amoretti, Alessandro Grazioli, Francesco Zanichelli, Valerio Senni, and Francesco Tiezzi. Towards a formal approach to mobile cloud computing. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 743–750. IEEE, 2014.

[41] M. Amoretti, A. Grazioli, V. Senni, F. Tiezzi, and F. Zanichelli. A formalized framework for mobile cloud computing. *Service Oriented Computing and Applications*, 2014. `doi:10.1007/s11761-014-0169-3`.

[42] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *Software Engineering, IEEE Transactions on*, 24(5):315–330, May 1998. `doi:10.1109/32.685256`.

[43] Oana Andrei and Hélene Kirchner. A higher-order graph calculus for autonomic computing. In *Graph theory, computational intelligence and thought*, pages 15–26. Springer, 2009.

[44] Mirko Viroli, Danilo Pianini, Sara Montagna, and Graeme Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 295–302. ACM, 2012.

[45] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The scel language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, July 2014. URL: `http://doi.acm.org/10.1145/2619998`, `doi:10.1145/2619998`.

[46] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. Modeling adaptation with a tuple-based coordination language. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1522–1527. ACM, 2012.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[48] M. Amoretti, A. Grazioli, and F. Zanichelli. An autonomic approach for p2p/cloud collaborative environments. *Peer-to-Peer Networking and Applications*, pages 1–16, 2015. URL: `http://dx.doi.org/10.1007/s12083-015-0367-6`, `doi:10.1007/s12083-015-0367-6`.

[49] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001. URL: `http://doi.acm.org/10.1145/964723.383071`, `doi:10.1145/964723.383071`.

[50] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin Heidelberg, 2002. URL: `http://dx.doi.org/10.1007/3-540-45748-8_5`, `doi:10.1007/3-540-45748-8_5`.

[51] Xinlei Wang, Wei Cheng, P. Mohapatra, and T. Abdelzaher. Enabling reputation and trust in privacy-preserving mobile sensing. *Mobile Computing, IEEE Transactions on*, 13(12):2777–2790, Dec 2014. `doi:10.1109/TMC.2013.150`.

[52] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982. URL: `http://www.sciencedirect.com/science/article/pii/0004370282900200`, `doi:http://dx.doi.org/10.1016/0004-3702(82)90020-0`.

[53] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010. `doi:http://doi.ieeecomputersociety.org/10.1109/MC.2010.98`.

[54] Huaming Wu, Qiushi Wang, and Katinka Wolter. Tradeoff between performance improvement and energy saving in mobile cloud offloading systems. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 728–732. IEEE, 2013.

[55] M. Amoretti, A. Grazioli, and F. Zanichelli. A modeling and simulation framework for mobile cloud computing. *Simulation Modelling Practice and Theory*, 2015. URL: `http://www.sciencedirect.com/science/article/pii/S1569190X15000799`, `doi:http://dx.doi.org/10.1016/j.simpat.2015.05.004`.

[56] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[57] Ioannis Moschakis, Helen D Karatza, et al. Performance and cost evaluation of gang scheduling in a cloud computing system with job migrations and starvation handling. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 418–423. IEEE, 2011.

[58] Michele Amoretti, Marco Picone, Francesco Zanichelli, and Giorgio Ferrari. Simulating mobile and distributed systems with deus and ns-3. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 107–114. IEEE, 2013.

[59] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, volume 14, 2010.

[60] Andrew Rice and Simon Hay. Decomposing power measurements for mobile devices. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 70–78. IEEE, 2010.

[61] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smart-

phone using kernel activity monitoring. In *USENIX Annual Technical Conference*, pages 387–400, 2012.

[62] Luca Ardito, Giuseppe Procaccianti, Marco Torchiano, and Giuseppe Migliore. Profiling power consumption on mobile devices. In *Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY), 2013 3rd International Conference on*, pages 101–106. IARIA, 2013.

[63] Luis Corral, Anton B Georgiev, Alberto Sillitti, and Giancarlo Succi. A method for characterizing energy consumption in android smartphones. In *Green and Sustainable Software (GREENS), 2013 2nd International Workshop on*, pages 38–45. IEEE, 2013.

[64] Xiaohan Ma, Zhigang Deng, Mian Dong, and Lin Zhong. Characterizing the performance and power consumption of 3d mobile games. *Computer*, 46(4):76–82, 2013.

[65] Michele Amoretti. A survey of peer-to-peer overlay schemes: effectiveness, efficiency and security. *Recent Patents on Computer Science*, 2(3):195–213, 2009.

[66] A. Montresor and L. Abeni. Cloudy weather for p2p, with a chance of gossip. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 250–259, Aug 2011. `doi:10.1109/P2P.2011.6038743`.

[67] Raymond Sweha, Vatche Ishakian, and Azer Bestavros. Angelcast: Cloud-based peer-assisted live streaming using optimized multi-tree construction. In *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, pages 191–202, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2155555.2155587`, `doi:10.1145/2155555.2155587`.

[68] James S Plank. T1: erasure codes for storage applications. In *Proc. of the 4th USENIX Conference on File and Storage Technologies*, pages 1–74, 2005.

[69] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *Peer-to-Peer Systems IV*, pages 226–239. Springer, 2005.

[70] Marco Martalò, Michele Amoretti, Marco Picone, and Gianluigi Ferrari. Sporadic decentralized resource maintenance for p2p distributed storage networks. *Journal of Parallel and Distributed Computing*, 74(2):2029–2038, 2014.

[71] Hanna Kavalionak, Emanuele Carlini, Laura Ricci, Alberto Montresor, and Massimo Coppola. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-to-Peer Networking and Applications*, 8(2):301–319, 2013.

[72] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1):173–187, 1999.

[73] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

Acknowledgements

As I am finally writing the final page of this thesis, I remember the most important facts that occurred during the last three years. It has been a great experience which allowed me to learn a lot and meet amazing people who influenced the way I see things, and opened my mind about the research and the progress of humanity.

First of all, I want to thank my supervisor, Prof. Francesco Zanichelli and Prof. Michele Amoretti, for supporting me during this long experience by providing suggestions, interesting ideas, and allowing me to go to a couple of good conferences. I hope we will continue to collaborate in the future, write more great papers and implement interesting projects. Also, I want to thank Dr. Marco Picone, Dr. Simone Cirani and Prof. Gianluigi Ferrari for our collaborations and the work opportunities they found for me. I also want to thank everybody who helped me with my research activity: Luca Barili, Giada Cilloni, Andrea Giannetti, Federico Magliani, Marco Magnani, Manuel Sequino. A big thanks also goes to the guys at DNAPhone S.r.l. and Your App For S.r.l. for the working opportunities they gave me. Thanks to my PhD colleagues for the suggestions and chats we had: Laura Belli, Andrea Gorrieri, Fabio Oleari, Antonio Prioletti.

Of course, a huge thanks goes to my whole family, but most of all, to my parents and grandparents. I know that supporting the decision of pursuing a PhD was not easy, but I am really grateful for having let me do it (with only a quite acceptable amount of complaints). A big thanks goes to Sónia, for everything she gives me everyday, to Kevin, for his suggestions as an ex-PhD student, to Marta and Elsa for the laughs, and to Huaming Wu and Marie-Paule Uwase for our interesting chats about how the research is in different countries. Finally, I want to thank my colleague Giacomo Brambilla, who turned out to be a priceless friend and gave me countless ideas, suggestions and laughs, both during the work and break times. Who would say that I would meet a classic *point & click* adventure games fan in a research lab?

Thanks everyone!
Alessandro