Code Migration in Mobile Clouds with the NAM4J Middleware

Alessandro Grazioli, Marco Picone and Francesco Zanichelli
Information Engineering Department
Università degli Studi di Parma
Parma, Italy
alessandro.grazioli1@studenti.unipr.it
{marco.picone,francesco.zanichelli}@unipr.it

Michele Amoretti
Interdept. Centre SITEIA.PARMA
Università degli Studi di Parma
Parma, Italy
michele.amoretti@unipr.it

Abstract—Mobile Cloud Computing (MCC) is a model for transparent elastic augmentation of mobile device capabilities via ubiquitous wireless access to cloud storage and computing resources. The main purpose of MCC is to exploit the context-aware dynamic offload of demanding mobile applications to the Cloud, in order to improve their performance while saving energy and extending battery lifetime of devices. In this paper we extend a pre-existing MCC taxonomy, and we illustrate how the autonomic approach enabled by the open source NAM4J middleware with code migration support can effectively address MCC requirements. We recall the architecture of NAM4J and show its capabilities in the context of an Ambient Intelligence (AmI) MCC application for the Android platform.

I. INTRODUCTION

Compared to today’s PC and server platforms, mobile devices struggle to cope with compute-intensive applications such as complex media processing and large-scale data management and mining. Moreover, user expectations are increasing, for which mobile applications are getting more and more complex. A way to extend or augment the capabilities of resource-constrained devices is Cloud Computing (CC), a new computing paradigm that allows users to temporary utilize computing, storage, bandwidth, application resources over the network, supplied as a service by the cloud-provider at possibly one or more levels of abstraction [1].

Mobile Cloud Computing (MCC) is a paradigm for transparent elastic augmentation of mobile device capabilities via ubiquitous wireless access to cloud storage and computing resources [2]. The main purpose of MCC is to exploit the context-aware dynamic offload of demanding mobile applications to the Cloud, in order to improve their performance while saving energy and extending battery lifetime of devices. On the other hand, to move data and processing outside of the mobile device enables new types of applications such as context-aware mobile social networks. From a smartphone perspective in particular, MCC opens up possibilities for a new class of applications by leveraging handset centric features and network related information, such as GPS and/or cell-based location information.

Current mobile applications are statically partitioned, i.e. most of the execution happens either on the device or on back-end systems. However, mobile clients could face wide variations and rapid changes in network conditions (e.g. vertical handover [3]) and local resource availability when accessing remote data and services. As a result, one partitioning model does not satisfy all application types and devices [4]. In order to enable applications and systems to continuously operate in such dynamic environments, MCC applications must be context-aware, i.e. they must react with dynamical adjusting of the computing functionality between the mobile device and cloud depending on circumstances.

In this paper we extend a pre-existing classification of MCC, with reference to the state of the art, using the granularity of the offloading process (ranging from device cloning to application partitioning and migration), and the degree of involvement of the Cloud, as a metrics. Moreover, we illustrate how the open source NAM4J middleware can enable the development of applications adhering to the MCC paradigm. NAM4J is based on the Networked Autonomic Machine (NAM) formalism, which allows to model a hardware/software system able to communicate with other NAMs, execute a number of functional modules that may provide/consume services or context events, dynamically deploy, undeploy and migrate functional modules and services. The adoption of autonomic policies to support MCC has never been investigated in depth, so far. This paper provides one of the first concrete contributions, in this context. Both NAM and NAM4J are projects that we have developed in the last two years. In this paper, for the first time, we illustrate the code migration mechanisms and their application to the MCC context.

The paper is organized as follows. In section II we present the MCC classification. In section III we illustrate the architecture of NAM4J. In section IV we describe an application example, in the context of Ambient Intelligence (AmI). Finally, in section V we conclude the paper with some ideas for possible future work.

II. MCC CLASSIFICATION

Many issues related to computation offloading have been investigated in the past decade. A recent article by Kumar et al. [6] provides an overview of the background, techniques, systems, and research areas for offloading computation. In particular, the main research focuses have been making offloading feasible, making offloading decisions, and developing offloading infrastructures. Despite the significant amount of research that has been performed, the offloading problem is still
challenging. As the number of connected devices — including smartphones, tablets, laptops, and sensors — grows, the demand for increased functionalities will continue. In the next few years, we will see pressing needs for personalized management of multimedia data. Due to several constraints, including form factor and power consumption, mobile computing speeds will not grow as fast as the growth in data, and applications computational requirements. Offloading computation to the Cloud is a natural solution to this problem.

We propose an extension to the MCC classification proposed by by Kovachev and Klamma [7]. In their survey paper, they overlooked to consider the mediating role the Cloud may play when heterogenous mobile devices get in touch and agree to cooperate for the execution of complex applications. Thus, we consider four reference approaches (illustrated in Fig.1), which differ in the granularity of the offloading process (ranging from device cloning to application partitioning and migration), and in the degree of involvement of the Cloud.

With Augmented Execution, some or all of the tasks are offloaded from the mobile device to the Cloud, where a cloned system image of the device is running [8] [9]. The results from the augmented execution are reintegrated upon completion. This approach for offloading intensive computations employs loosely synchronized virtualized or emulated replicas of the mobile device in the Cloud. Thus, it provides the illusion that the mobile user has a more powerful, feature-rich device than actually has in reality, and that the application developer is programming such a powerful device without having to manually partition the application or provision proxies.

In a different context, Elastically Partitioned Applications can improve their performance by delegating part of the application to remote execution on a resource-rich cloud infrastructure. Cuervo et al. propose to annotate while programming which methods can be offloaded for remote execution [10]. The profiling information, network connectivity measurements, bandwidth and latency estimations are used as input parameters for an optimization problem which is periodically solved to give a decision which methods and when should be offloaded. However, we think that such an approach would hardly scale in a pervasive computing environment, where too many variables could be involved. Moreover, such a fine-grained offloading of mobile code to the cloud infrastructure could be unnecessary. Rather than having complex applications with tens of loadable methods, we claim that it would be better to have more modular applications, with the possibility to offload modules based on simple cost models.

A Spontaneous Mobile Cloud represents a group of mobile devices, connected by means of an infrastructure (WiFi, 3G, etc.) or in ad hoc mode, that serve as a cloud computing provider by exposing their computing resources to other mobile devices [11]. This type of peer-to-peer MCC applies to situations with no or weak connections to the Internet and large cloud providers. In that situation, the other peers mimic the unavailable real Cloud. Moreover, it enables computing communities in which users can collaboratively execute shared tasks.

Still unexplored, the Hybrid Mobile Cloud approach consists of two or more mobile devices which collaborate with the support of a remote Cloud. Suppose that Bob wants to offload a task to the mobile node of Alice, but the latter has a different hardware and operating system, for which the code cannot be directly migrated from Bob's device to Alice's one. Thus, the latter gets the code (in a suitable bundle) from the Cloud. If data and execution state are necessary, Bob directly sends them to Alice, with a direct device-to-device communication.

These four approaches can be compared by placing them in a planar space, whose axes represent the Cloud involvement, and the grain level of the offloading, as illustrated in figure 2. The Augmented Execution approach is characterized by coarse grain offloading and high Cloud usage. The other three approaches are characterized by fine grain offloading and a different degree of Cloud usage — from zero, in the Spontaneous Mobile Cloud approach, to increasing levels in the Hybrid Mobile Cloud and Elastically partitioned Applications approaches.

![Planar space of MCC approaches.](image)

**III. NAM4J Middleware**

The adoption of autonomic policies to support MCC has the advantage of alleviating the mobile user from manually starting/stopping applications, or application modules, when their execution becomes too much demanding in terms of local resources. The approach we propose has also another desirable feature, *i.e.*, it supports code migration, thus allowing the dynamic upgrading/downgrading of mobile applications,
without the necessity of hardcoding the autonomic policies. This is allowed by the NAM4J middleware, our open source\textsuperscript{1}, Java-based implementation of the NAM framework \cite{forsgren2007}. In this section we illustrate the structure of the middleware, with reference to the main concepts of the framework.

Formally, a NAM is a tuple $\langle R, F \rangle$, where $R$ is a set of physical resources (e.g. processing units, storage devices, download/upload bandwidth), and $F$ is a set of functional modules. Each functional module $f \in F$ plays one or more of the following roles: context provider (CP), context consumer (CC), service provider (SP), service consumer (SC).

The NetworkedAutonomicMachine class — which is abstract, to force its extension in specialized NAM classes — provides methods for the runtime deployment and undeployment of functional modules, as well as the methods for the remote discovery and migration of functional modules, which are described in detail below. Associating a NAM with a functional module (by means of the addFunctionalModule method) does not imply the automatic execution of the latter, in general. The abstract class FunctionalModule exposes a method called execute, which should be invoked in order to start the functional module itself.

A functional module is defined as:

$$ f = \langle S_I, S_r, C_{in}, C_{out}, OB, FP, SMP \rangle $$

where $S_I$ is the set of provided services, $S_r$ is the set of services the module can consume, $C_{in}$ is the set of consumed context events, $C_{out}$ is the set of provided context events, $OB$ is the set of objectives, $FP$ is the set of functional policies, $SMP$ is the set of self-management policies according to which the functional module performs self-configuration, self-healing, self-optimization and self-protecting. Functional policies are rules, algorithms, evolutionary plans, etc. the functional module adopts in order to meet its objectives. The policy specification language has been described in a previous work \cite{forsgren2007}. Its implementation is work in progress and will be described in a future work.

### A. Context Awareness

Context events are information pieces that describe changes in the environment. A CP is a software entity placed between sensors or other CPs, and CCs. Context awareness is usually implemented following the publish / subscribe paradigm, for which CCs express interest in one or more context event classes, and only receive context events that are of interest, without knowledge of what, if any, CPs there are. Context events may be published either periodically or sporadically. A context event is expressed in the form "subject - action - object - location" and has a timestamp (indicating when it has been generated), thus responding to the following questions: who? what? where? when? To this purpose, the ContextEvent class exposes methods for getting and setting name, timestamp, temporal validity, producer, subject, action, object and location.

An example of context event is illustrated in box 1. The TemperatureNotification class extends ContextEvent, while Temperature and Room extends Parameter. For this type of context event, the timestamp should represent the exact time of the measurement (of the actual room temperature, in the example).

#### 1. Example of context event

```java
Temperature temperature = new Temperature();
temperature.setValue("22");
TemperatureNotification tempNotif = new TemperatureNotification();
tempNotif.setSubject(temperature);
tempNotif.setLocation(room);
tempNotif.setTimestamp(new Date());
Date timestamp = new Date();
DateFormat df = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
tempNotif.setTimestamp(df.format(timestamp));
```

In other situations, the timestamp could be simply a mean for ordering events. For example, the context events "Bob cleaned - the room - at $t_1$" and "Bob cooking - at $t_2$", with $t_1 < t_2$, say that Bob cleaned the room and then he cooked (not vice versa), but they do not specify how much time the two actions did take. When receiving one or more context events in $C_{in}$, a functional module may react by publishing a set of context events in $C_{out}$, by executing services in $S_I$, or by calling services in $S_r$.

In our current prototypes, context events are serialized into JSON messages\textsuperscript{2}, using the GSON library\textsuperscript{3}. JSON (JavaScript Object Notation) is a lightweight data-interchange format, easy for humans to read and write, and easy for machines to parse and generate. Basically, there is a JSON library for any programming language, mapping their specific data structures (objects, records, hash tables, arrays, etc.) into JSON data structures — which are collection of name/value pairs.

### B. Tasks and Services

In the NAM perspective and terminology, a task is a high-level, user-oriented term associated with the user’s requirements, while a service is a low-level, system-oriented term associated with system functionalities. A service may be atomic or composite, and characterizes both software and hardware functionalities in smart environments. Indeed, in the concept of service we include not only information services (such as multimedia data processing), but also device services (such as climate control). A service consists in a unit of work executed by a service provider to achieve the results desired by a service consumer. A SP is a software entity that exposes a set of services to SCs, and executes such services upon request. It is remarkable that context events can be retrieved on-demand, by invoking appropriate information services.

The Service class, which can be used for the implementation of both atomic and composite services, implements the definition of service as a tuple

$$ s = \langle I, O, P, E \rangle $$

where $I$ is a set of input parameters, and $O$ is a set of output parameters. Each $I/O$ parameter has a type, i.e. a class (still using the ontological terminology). It is important that service consumers and service providers share the same

\textsuperscript{1}http://code.google.com/p/nam4j/
\textsuperscript{2}http://www.json.org/
\textsuperscript{3}http://code.google.com/p/google-gson/
domain ontologies in order to have a common understanding of shared services. Semantic descriptions of services are used to organize service advertisements in centralized or distributed repositories, allowing to efficiently retrieve and use services in the network. $P$ and $E$ are the precondition and effect sets, respectively. Such optional parameters are expressed in the form of logical conditions which can assume the true or false value. Preconditions must be verified in order to invoke the service, while an execution effect may become a precondition for the successive invocation in a composition scenario. The IOPE approach is adopted, for example, in the OWL-based Web service ontology called OWL-S.

An example of service is illustrated in box 2. When the service instance is created, its run() method is executed in a separate thread. To reduce the overhead, a threadpool\(^5\) could be used.

2. Example of service

```java
public class TestService extends Service {
    public TestService(String id, String name) {
        this.setId(id);
        this.setName(name);
        Thread t = new Thread(
            new TestServiceRunnable(this),
            "TestService thread");
        t.start();
    }
    public void run() {
        // ....
    }
}
class TestServiceRunnable implements Runnable {
    private TestService ts = null;
    public TestServiceRunnable(TestService ts) {
        this.ts = ts;
    }
    public void run() {
        ts.run();
    }
}
```

C. Migration

Kumar et al. [2] have shown that cloud-based energy saving on mobile devices is achieved when (a) the ratio between the amount of data $D$ exchanged between the Cloud and the mobile system and the network bandwidth $B$ is sufficiently small compared with the ratio between the number of instructions $C$ required by the computation and the execution rate $M$ of the mobile system, and (b) $S/M$ is sufficiently large (where $S$ is the execution rate of the Cloud). Deciding the optimal $D$ is a challenging task, but autonomic techniques may support it effectively.

NAMs are allowed to dynamically reconfigure their structure, by adding new functional modules or services, or discarding those that are no more necessary, according to appropriate self-optimization policies. The migration of functional modules and services is defined as

$$m_f = \langle NAM_u, NAM_d, R, f, D, E \rangle$$

$$m_s = \langle NAM_u, NAM_d, R, s, D, E \rangle$$

where $f \in F$ is a functional module, $s \in S$ is a service, $NAM_u$ is the uploader, $NAM_d$ is the downloader, $D$ is the set of input data for the computation, and $E$ is the execution state (both strong and weak mobility are supported). A migration can start only if the resources of the destination host fit with $R$, the set of resources (e.g. running environment, libraries, CPU, RAM, etc.) required to run the functional module or the service to be migrated. A migration process can be application-specific, or transversal to the applications running on top of the NAM. In the latter case, migration is one of the self-regulatory processes of the NAM.

To offload a functional module, first of all it is necessary to find a remote NAM, by asking a bootstrap server, or using a decentralized discovery service — such as the one provided by the $f_{Chard}$ module described in section IV. Then, it is necessary to check if one of the known remote NAMs has enough resources to run the functional module. The check-NAMAvailability() method is invoked, with the following input parameters:

- a list of descriptors of required resources, i.e. instances of the ResourceDescriptor class
- the identifier of the remote NAM

When called, the availability check method establishes a TCP connection with the remote NAM (which runs a threadpool for managing incoming availability requests), to send the list of required resource descriptors (serialized to a JSON message). If the remote NAM has the required resources (most probably, if the remote NAM runs in the Cloud), it replies with an OK message.

Once a suitable NAM has been found, for migrating the functional module, the NAM that started the check process executes the migrateFunctionalModule() method (shortly, $m_f$ method), with the following input parameters:

- the name of the main class of the functional module (extending NAM4J’s base class FunctionalModule)
- a list of required jars (at least one, containing the classes of the functional module to be migrated)
- the identifier of the remote NAM

When called, the $m_f$ method establishes a TCP connection with the remote NAM (which runs a threadpool for managing incoming migrations), to send the name of the main class of the functional module as well as the required jars. Once the transmission completes, the remote NAM dynamically adds the received jars to its classpath and instantiates the functional module.

To offload a service, the procedure is very similar. The only difference is in the signature of the migrateService() method, which includes also — as input parameter — the identifier of the remote functional module that will host the service.

The inverse migration process, for which a NAM requests and obtains a functional module or a service from another NAM, is supported by the activateMigration(), findRemoteFM(String serviceName) and findRemoteService(String serviceName) methods. To serve incoming requests, each NAM has a specific threadpool, which is different with respect to the one used to serve offloading requests.

\(^4\)http://www.daml.org/services/owl-s/1.0/owl-s.html

\(^5\)http://www.yoda.arachsys.com/csharp/threads/threadpool.shtml
The Java code for dynamically adding jars to the classpath is reported in box 3. For the Android platform, the DexClassLoader must be used instead of ClassLoader, and each jar file must contain a classed.dex file. Importantly, to use these functionalities provided by NAM4J, it is not necessary to have rooted or jailbroken versions of the Android devices. Thus, the number of potential users is very large.

3. Dynamic jar loading
   
   public static void addURL(URL u)  
   throws IOException {  
   URLClassLoader sysloader = (URLClassLoader)  
   ClassLoader.getSystemClassLoader();  
   Class<?> sysclass = URLClassLoader.class;  
   try {  
   Method method =  
   sysclass.getDeclaredMethod("addURL",  
   parameters);  
   method.setAccessible(true);  
   method.invoke(sysloader,new Object[]{u});  
   }  
   catch (Throwable t) {  
   ...  
   }

   To add security features, we are studying two different approaches. Code signing is a useful process to protect code when transferred, both from passive (i.e. eavesdropping and traffic analysis) and active attacks (i.e. message modification, deletion, forging); code is digitally signed by the provider in order to assure strong authentication and integrity to the consumer. Another approach is to use reflection (supported by Java, and in particular by JVM and Android Dalvik) to inspect the downloaded package, and also to test it in a controlled environment, to check if its behavior is the expected one.

IV. APPLICATION EXAMPLE

Consider an ambient intelligence (AmI) system where the user mobile device runs an application which collects context events from sensors placed in the environment and process them to produce aggregated monitoring information and control messages for the actuators. As the required functional modules may have a significant impact on system autonomy, whenever some battery charge needs to be preserved, the application could offload one or more functional modules, as well as the underlying ontologies and related code mappings (which usually ships as linked libraries), to another user device (e.g., a relative, or a colleague, depending on the context) or to the Cloud. Migration mechanisms can also be effective to enrich the capabilities of a node.

We have prototyped such an AmI scenario for the Java/Android platforms, by defining and deploying several NAMs, both fixed and mobile, each endowed with different functional modules and services. With reference to Fig. 3, the mobile NAMs $n_{m}^{1}$ and $n_{m}^{2}$, and the fixed NAMs $n_{f}^{1}$ and $n_{f}^{2}$ are equipped with a functional module $f_{Chord}$, which allows them to connect to a peer-to-peer network based on the Chord protocol [12]. The mobile NAM $n_{m}^{m}$ represents a newcomer node initially lacking $f_{Chord}$ and entering the same physical environment of $n_{f}^{1}$, so that the two NAMs can get in touch and communicate, for example over Bluetooth. Because of that, the mobile NAM $n_{m}^{m}$ can obtain $f_{Chord}$ from $n_{m}^{1}$, by means of NAM4J migration mechanism, and starts executing it, thus joining the existing Chord ring (as shown in Fig. 4).

Fig. 3. The mobile NAM $n_{m}^{m}$ obtains the $f_{Chord}$ bundle from $n_{m}^{1}$.

Fig. 4. The mobile NAM $n_{m}^{m}$ joins the Chord ring.

Belonging to the Chord overlay network provides several advantages. The most appealing feature is that the number of nodes that must be contacted to publish or find a data item is only $O(\log N)$, on average, where $N$ is the number of connected nodes which, unlike our particular example, can be arbitrarily large. Moreover, if a data item has been published, its retrieval is almost certain, due to the deterministic nature and robustness of Chord’s publish/lookup protocol [12]. Each peer owns a routing table which targets a very limited number of other peers (chosen according to a deterministic strategy). The routing table serves for publishing information about shared resources, evenly distributed among peers.

The NAM $n_{m}^{m}$ is provided with a functional module $f_{sense}$, which is a consumer of context events, but also a service provider. Indeed, the $f_{sense}$ is equipped with a service $s_{state}$ which returns the state of the environment, as observed by $f_{sense}$ (which may monitor temperature, humidity, CO level, presence, etc.). If $n_{m}^{m}$ has a reason to leave the environment, an autonomic policy may trigger the assessment of $n_{m}^{m}$ availability to receive and execute $f_{sense}$ (and $s_{state}$). If $n_{m}^{m}$ accepts, the bundles of $f_{sense}$ and $s_{state}$ are migrated from $n_{m}^{m}$ to $n_{m}^{m}$. Once $f_{sense}$ is deployed and running on $n_{m}^{m}$, $n_{m}^{m}$ will still be able to invoke $s_{state}$ from a remote location to monitor its previous physical environment in mobility.

---

6http://docs4words.blogspot.com/2013/03/android-runtime-class-loading.html
Due to an unexpected shortage of local resources, an energy saving autonomic policy may drive NAM $n_3^m$ to look for an external Cloud storage service, to store data sensed in its physical environment. The descriptor of a suitable service $d(s_{\text{storage}})$ was published in the Chord ring, and is under the responsibility of $n_1^f$. As shown in Fig. 6, $n_3^m$ sends a lookup request, and once the service descriptor has been found in $n_1^f$, $n_3^m$ retrieves it, and starts using the cloud service.

The proposed example gives some insights on how the MCC paradigm, particularly when coupled with autonomic policies and code migration capabilities, can improve the effectiveness and efficiency of complex distributed applications integrating mobile nodes and Cloud services within dynamic and smart environments.

V. CONCLUSION

In this article we have proposed a classification of MCC, with four approaches which differ in the granularity of the offloading process (ranging from device cloning to application partitioning and migration), and in the degree of involvement of the Cloud. Then, we have illustrated the NAM4J-based code mobility approach which exploits self-management policies for the energy-aware and efficiency-oriented offloading of code bundles from mobile devices to the Cloud.

Regarding future work, we plan to experiment pervasive applications based on the Hybrid Mobile Cloud approach, focusing on Android and iPhone platforms. An interesting research topic is the performance characterization of MCC architectures and its exploitation in the design of improved execution offloading algorithms. In this context, we plan to perform an insightful analysis of the execution of Hybrid Mobile Cloud applications. Moreover, regarding the description and reasoning about resources and services in NAM4J, we plan to investigate alternative technologies to OWL/OWL-S — in particular, we are interested in the full adoption of the JSON technology, which is lighter than XML and better copes with constrained devices.

REFERENCES