SOCA (2015) 9:229–248
DOI 10.1007/s11761-014-0169-3

SPECIAL ISSUE PAPER

A formalized framework for mobile cloud computing

Michele Amoretti · Alessandro Grazioli · Valerio Senni · Francesco Tiezzi · Francesco Zanichelli

Received: 19 March 2014 / Revised: 4 October 2014 / Accepted: 24 October 2014 / Published online: 6 November 2014

Abstract Mobile Cloud Computing (MCC) is an emerging paradigm to transparently provide support for demanding tasks on resource-constrained mobile devices by relying on the integration with remote cloud services. Research in this field is tackling the multiple conceptual and technical challenges (e.g., how and when to offload) that are hindering the full realization of MCC. The Networked Autonomic Machine (NAM) framework is a tool that supports and facilitates the design networks of hardware and software autonomic entities, providing or consuming services or resources. Such a framework can be applied, in particular, to MCC scenarios. In this paper, we focus on NAM's features related to the key aspects of MCC, in particular those concerning code mobility capabilities and autonomic offloading strategies. Our first contribution is the definition of a set of high-level actions supporting MCC. The second contribution is the proposal of a formal semantics for those actions, which provides the core NAM features with a precise formal characterization. Thus, the third contribution is the further development of the NAM conceptual framework and, in particular, the partial re-engineering of the related Java middleware. We show the effectiveness of the revised middleware by discussing the implementation of a Global Ambient Intelligence case study.

This work has been partially sponsored by the EU projects ASCENS (257414) and QUANTICOL (600708), and by the Italian MIUR PRIN project CINA (2010LHT4KM).

M. Amoretti (✉) · A. Grazioli · F. Zanichelli
Department of Information Engineering,
University of Parma, Parma, Italy
e-mail: michele.amoretti@unipr.it

V. Senni · F. Tiezzi
IMT Advanced Studies Lucca, Lucca, Italy

Keywords Mobile cloud computing · Programming abstractions · Formal semantics · Middleware design

1 Introduction

Mobile Cloud Computing (MCC) is an emerging paradigm for transparent elastic augmentation of mobile devices capabilities, exploiting ubiquitous wireless access to cloud storage and computing resources [25]. MCC aims at increasing the range of resource intensive tasks supported by mobile devices, with no or limited effects on their battery autonomy. While the ever increasing communication capabilities available in mobile devices make viable offloading computation and storage to remote services, several issues and challenges are hindering the full realization of MCC. 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, as well as privacy and security concerns related to storing user data on a remote cloud [15,16,24]. Moreover, as multiple offloading approaches are possible [24] depending on the task and context, autonomic computing techniques appear promising to increase the robustness and flexibility of MCC systems [10]. In particular, autonomic policies grounded on continuous resource and connectivity monitoring may help automate the context-aware selection and operation of offloading procedures.

The Networked Autonomic Machine (NAM) framework [3] is a general-purpose conceptual tool to describe distributed autonomic systems, and it is suitable for MCC systems, as it supports code and data mobility concepts. The Java implementation of a middleware based on NAM, called

NAM4J, has been recently enhanced with support for code mobility on mobile platforms. However, reasoning on MCC issues and optimizing relevant mechanisms within NAM are not an easy task, as the middleware contains too low-level details, while the conceptual framework is too abstract.

The aim of this paper was to provide the NAM framework with formal grounds, in order to fill the gap between its implementation and its conceptual definition. In particular, we focus on those aspects that are important for its adoption in MCC scenarios. To this aim, we propose an approach that synergistically combines foundational concepts from the Theoretical Computer Science field, e.g., process calculi, with practical issues from the Software Engineering field, which concern the implementation of the NAM4J middleware. Therefore, our goal is not to propose a theory that is an end-in-itself, but a formal approach that endeavors to have an impact on the actual MCC technology.

More specifically, we provide the NAM framework with a formalization in terms of a *transformational operational semantics* based on the Kernel Language for Agents Interaction and Mobility (KLAIM) [12]. Intuitively, such formalization associates a KLAIM term to any NAM construct. The use of KLAIM for this purpose is a natural choice, because it is a linguistic formalism specifically designed to model distributed systems consisting of several mobile components which interact through multiple distributed shared memories, called tuple spaces. Its primitives allow programs to distribute/retrieve *data* and *processes* to/from the nodes of a network, thus enabling data and code mobility.

In addition, we have revised the NAM conceptual framework by equipping it with *five different mobility primitives* that can be employed in the design and implementation of MCC applications. Indeed, our first step has been the identification of this set of high-level primitives to be used in NAM as first-class citizens, in order to allow developers to directly focus on MCC concerns rather than their low-level implementation. In fact, it is the duty of our formalization to express them in terms of traditional primitives for code mobility and coordination.

Moreover, driven by this formal treatment of the mobility features of NAM, we have partially re-engineered its NAM4J Java middleware, leading to an enhancement of its API supporting MMC features and a clearer definition of the development methodology. To experiment with the revised version of NAM4J and to demonstrate its effectiveness, we have implemented a practical illustrative example, where mobile devices are used to collect and elaborate (possibly relying on cloud resources) environmental data from local and external sensors.

To sum up, the formalization process contributed to: (1) the identification of a set of high-level MCC-oriented primitives; (2) the refinement of the NAM framework to include them and a formal characterization of its semantics; and (3) the ensuing re-engineering of the NAM4J middleware. Moreover, such a formalization effort paves the way for the verification of qualitative and quantitative properties of MCC systems, thus supporting a formal-based design of autonomic context-aware offloading strategies.

Structure of the paper. The remainder of this paper is structured as follows. Section 2 presents a simplified description of the NAM framework, tailored to our formalization purpose. Section 3 describes NAM at work on an illustrative example from the MCC domain. Section 4 outlines the main features of KLAIM, which are used in Sect. 5 to define a formal semantics of NAM. Section 6 shows how the NAM formalization has driven the re-engineering of the mobility functionalities of NAM4J and demonstrates its effectiveness by means of the implementation of the example introduced in Sect. 3. Section 7 describes related work regarding MCC, autonomic middleware, code migration, and their formalization. Section 8 reports our conclusions and describes future work. Finally, the complete KLAIM-based formalization, as well as related comments, is reported in the Appendix.

This work is an extended and revisited version of our former development introduced in [1]. This version of the paper (1) provides a complete presentation of the KLAIM-based formalization, (2) illustrates the re-engineering effort of the middleware mobility actions, (3) shows the NAM approach, in particular the revised NAM4J middleware, at work on a practical MCC example on Global Ambient Intelligence.

For the ease of reading, Table 1 lists the acronyms used throughout the paper.

2 The NAM framework

A system of NAMs is a loosely connected network of hardware/software entities, which provide or consume services. In this paper, we focus on specific MCC features, such as data and code mobility, and thus we only consider the set of NAM concepts devoted to address them. Other NAM concepts, such as service composition and interface compatibility, are also relevant for a comprehensive description of MCC scenarios, but are not MCC-specific. Therefore, we decided to omit them, in order to focalize on mobility aspects. We plan to include also those aspects in a future extension of the NAM framework formalization.

In a NAM network, each device can host one or more NAMs. Roughly, a NAM is a container of data and computational entities. Both *application data* and *awareness data* are considered; the former is used for enabling the progress of the NAM's computation, while the latter provides information about the environment in which the NAM is running (e.g., sensor readings and context events) or about the status of the NAM itself. Computational entities, instead, are ser-

Table 1 List of acronyms

BH = Back action Handler
CH = Copy action Handler
Disp = Dispatcher
DST = Destination
GAmI = Global Ambient Intelligence
GH = Go action Handler
LPH = Local Policy Handler
LSH = Local Service Handler
KLAIM = Kernel Language for Agents Interaction and Mobility
MCC = Mobile Cloud Computing
MH = Mobility Handler
MiH = Migrate action Handler
NAM = Networked Autonomic Machine
NAM4J = NAM for Java
OH = Offload action Handler
PH = Policy Handler
PMH = Policy and Mobility Handler
RPH = Remote Policy Handler
RSH = Remote Service Handler
SH = Service Handler
SRC = Source

vice threads exploiting functionalities provided by libraries called *functional modules*.

More formally, a NAM is represented as a tuple $nam = \langle nid, \mathcal{R}, \mathcal{F}, \mathcal{P} \rangle$, where nid is the NAM identifier, $\mathcal{R}$ is a set of physical *resources*, $\mathcal{F} = \{f_1, \ldots, f_m\}$ is a set of functional modules, and $\mathcal{P}$ is a set of NAM (self-management) policies. Resources are, for example, CPU cycles, storage space, network interfaces, etc. Each NAM is allowed to directly access its own resources. Instead, remote resources (of another NAM) are not directly accessible. Actually, a NAM can only interact with the services exposed by other NAMs. For this reason, the formalization described in this work mostly focuses on services. We do not consider data as a resource, and we assume it is always stored within functional modules and moved accordingly. More general models including NAM data outside functional modules are out of the scope and purpose of this paper, although we do not envisage any issue in extending our formalization in such a direction. The state of a NAM consists of the sets $\mathcal{R}$, representing available resources, and $\mathcal{F}$ describing functional modules that currently reside on it. Autonomic policies are a crucial means to support MCC, since 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 an Event-Condition-Action rule of the form (ev, co, act): the occurrence of an event ev triggers the evaluation of the corresponding condition co and, in case of positive evaluation, the action act is executed.

A functional module is a specialized module represented as a tuple $f = \langle fid, \mathcal{S}, \mathcal{P}_f, \mathcal{D}, \mathcal{T} \rangle$, where *fid* is the functional module identifier, $\mathcal{S}$ is a set of bindings from service names to methods of f implementing them, $\mathcal{P}_f$ is a tuple containing functional policies of the module (i.e., policies that define module-specific actions to be taken when particular conditions hold), $\mathcal{D}$ is a set of data available to the module, and $\mathcal{T}$ is a set of threads currently run by the module itself. We consider a *service* as an entry point for a functional module, which has the role of aggregating functions and data to provide computational tasks. In other words, 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 (via bindings in $\mathcal{S}$) the corresponding local/remote method and subsequently creates a thread implementing it. Events are another form of entry points, but they differ from services since a service request triggers a thread execution, while 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. Specifically, functional module policies $\mathcal{P}_f = \langle \mathcal{P}_o, \mathcal{P}_l, \mathcal{P}_r \rangle$ are structured in three parts: $\mathcal{P}_o$ are the *on-site* policies, active when the module is not offloaded, while $\mathcal{P}_l, \mathcal{P}_r$ are the policies activated in the *local* and *remote* NAMs, respectively, when the module is offloaded. The need of having local and remote policies in offloading is motivated by the need of evaluating events both locally and remotely. An example of local event is the detection of decreasing connection quality, triggering the recall of a remote module. Similarly, a remote event can arise on lack of resources, triggering the decision of sending the module back to the owner.

2.1 Mobility actions

Mobility is a fundamental aspect of NAM networks, since it allows a dynamic reconfiguration of the system by moving functional modules from NAM to NAM. We allow for five different mobility actions: offload, back, go, migrate, and copy. Figure 1 summarizes the four scenarios where these actions can be used.

In the first scenario, nam_1 is running short of resources (such as battery or cpu) so it decides, according to its internal policies, to move the code of functional module f to nam_2 through an offload action. As an effect of this action, the resource-consuming elements of f (i.e., data $\mathcal{D}$ and running threads $\mathcal{T}$) are moved to nam_2 and are regulated by specific policies $\mathcal{P}_r$ (while, from now on, policies $\mathcal{P}_l$ are activated and enforced locally to nam_1). Therefore, f stops consuming

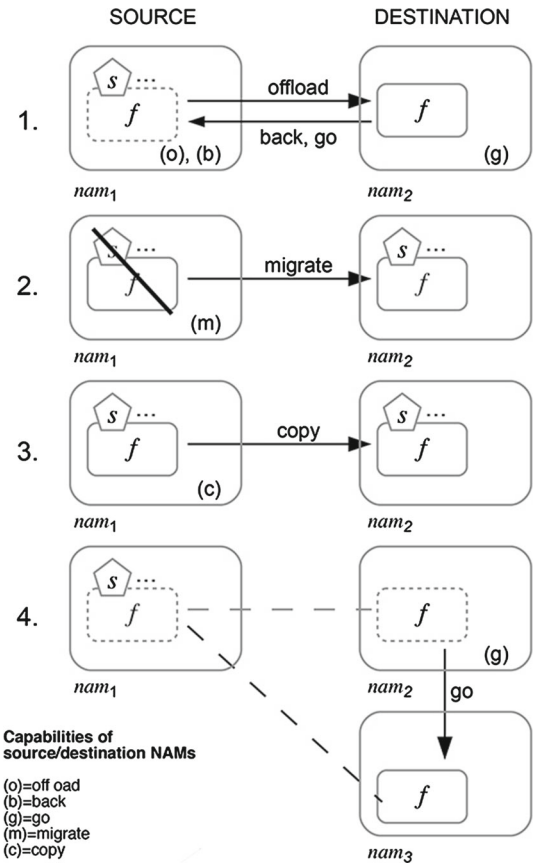

Fig. 1 Allowed mobility actions

resources of the source and starts consuming the ones of the destination. The entry points of f (i.e., the services specified in $\mathcal{S}$) are, instead, left on nam_1. This choice is motivated by the need of full offloading transparency, with respect to local and remote modules that use services of f. This operation requires the service bindings $\mathcal{S}$ on nam_1 to be modified to redirect service requests on nam_2. If necessary, nam_1 can request to terminate the offloading of f by executing a back action, which moves back the functional module f to nam_1 and updates $\mathcal{S}$ and active policies consistently. Finally, in the case nam_2 decides it cannot provide hosting for f any longer (e.g., nam_2 is a cloud service and nam_1 is running out of credit), it can execute a go action which, again, moves back the functional module f to nam_1.

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

In the third scenario, we consider events such as downloading applications or libraries. After a request of 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. This action modifies the set $\mathcal{F}_2$ of functional modules on nam_2.

Finally, in the fourth scenario, we consider operations that move offloaded modules. A typical case can be the need of moving an offloaded module from a NAM to another to perform load-balancing. In the figure, nam_2 hosts a module offloaded by nam_1 and decides it cannot offer hosting any more. Thus it moves f to nam_3 through a go action. This operation moves all elements of f from nam_2 to nam_3 and updates $\mathcal{S}$ on nam_1 (the update of these bindings is represented in Fig. 1 by the dashed lines).

Note that, in actions back, go, migrate, and offload, the execution of threads $\mathcal{T}$ of the module f is suspended and, then, recovered in the remote location. Similarly, local data $\mathcal{D}$ of the module are moved to the remote location. On the contrary, in a copy action, we expect f has no track of previous execution on nam_1. Therefore, the sets $\mathcal{D}$ of data and $\mathcal{T}$ of threads are set to be empty in nam_2.

A mobility action can be executed by a NAM on a local functional module, for actions copy, go, migrate, and offload, and on a remote functional module, for action back.

Notably, we currently do not allow to move services, unless the whole module is moved, since we do not envisage any benefit in the considered MCC application scenarios. Anyway, moving services would be a much lighter operation, because it consists essentially in moving/copying just the service name and updating the corresponding bindings.

3 Global Ambient Intelligence example

In this section, we show a simple, although quite realistic, MCC illustrative example, based on the NAM framework, described in Sect. 2 and formalized in Sect. 5. The aim is to clarify the role of mobility actions and, in particular, how policies permit to separate the decision-support logic from the code implementing mobility actions.

Let us consider a mobile sensing application allowing to retrieve and display different types of sensed information. Users are enabled to access context events and services everywhere, at home or outside, in mobility. Mobile devices carried by users can be raw data sources, but also aggregated information producers, which combine and process low-level data coming from sensors distributed in the environment. Indeed, the widespread and ubiquitous nature of mobile devices makes them attractive as providers of information collected from their rich equipment of sensors (camera, microphone, GPS, etc.), and also from external sensors (placed on people, or in the environment). We refer to such a mobile sensing

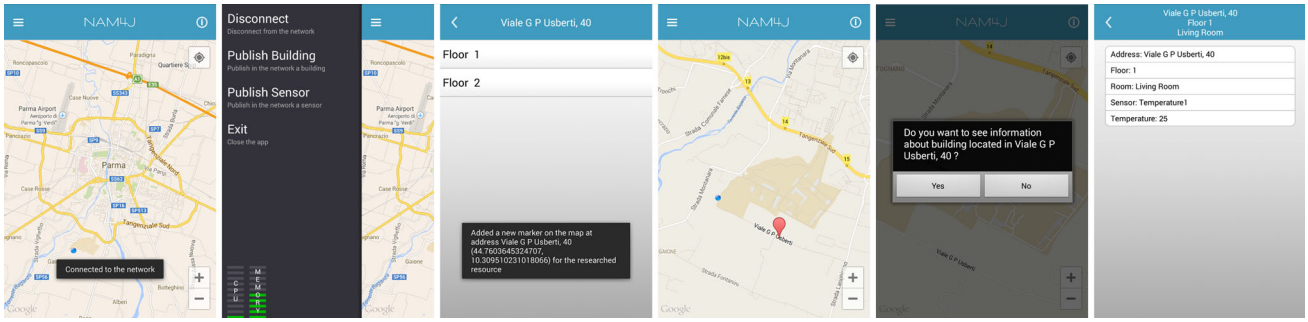

Fig. 2 Screenshots of the GAmI application. The user is allowed to publish building descriptions, including their sensor equipments. Moreover, the user can search for deployed sensors and obtain periodic updates of their measured values

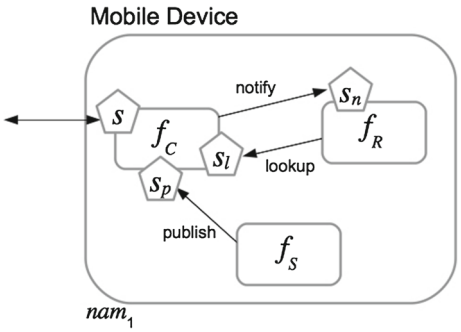

Fig. 3 The structure of the NAM of the GAmI application

approach as Global Ambient Intelligence (GAmI). A practical use of the mobile sensing application could be related to temperature monitoring and control, integrated with activity recognition to improve the comfort of the user.

In a previous work [3], we illustrated an implementation of such a GAmI application (see Fig. 2), using the basic features of the NAM framework. As a step forward, now, we illustrate the benefits of the MCC actions formalized in this paper. The envisioned distributed GAmI application is a NAM network, where each NAM can be provided with the following functional modules (see also Fig. 3):

- ChordFunctionalModule (f_C)—it allows the NAM to participate in a Chord overlay network [33], with Publish and Lookup services (s_p and s_l, respectively); Chord is characterized by an appealing $O(\log N)$ information lookup performance, being N the number of nodes in the network, thus allowing for highly scalable decentralized applications;
- SensorFunctionalModule (f_S)—it uses the Publish service of f_C to share context events related to sensed information;
- ReasonerFunctionalModule (f_R)—it uses the Lookup service of f_C to get context events of interest; to allow f_C also to behave in a proactive way, f_R exposes a Notify service (s_n) that is called by f_C when relevant context events are produced either by f_S, or by remote NAMs.

Among these functional modules, f_R is the most demanding in terms of CPU cycles, while f_C 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 its local sensors. To reduce the energy consumption, the best candidate for being offloaded or migrated is f_R, as f_C must stay on the mobile device for connectivity reasons, and f_S can reduce its operation rate. Instead, f_R is always active, but it must not necessarily run on the mobile device, to provide aggregated sensing information to the user. It is worth noting that copying f_C to other NAMs allows to increase the size of the Chord overlay network, which may be useful to balance the communication and information storage workload.

Thus, we consider an *Augmented Execution* scenario [24] in which a mobile device, hosting the GAmI NAM, is running short of a certain resource (e.g., battery power or CPU cycles), while f_R is performing its demanding task. In general, an autonomic application with MCC support (e.g., a subscription to a cloud service) should react accordingly to the situation, so that its 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 service and to copy f_C for execution on another mobile node.

It is crucial to identify the responsibility of such decisions, the mechanisms to enact them, as well as the responsibility of related mobility operations (e.g., back action). Regarding the offloading decision, a reasonable solution within the NAM framework is to rely on self-management policies. These are entitled of monitoring events related to the state of the device, in order to preserve a given quality of service or safety conditions. Let us now start considering the role of policies of the ReasonerFunctionalModule f_R in our specific example. Policies of f_R are $\langle \mathcal{P}_o, \mathcal{P}_l, \mathcal{P}_r \rangle$, where:

$$\mathcal{P}_o = \{(\text{cpuLoadUpdate}, load > 70\%, \text{offload}(fid)), (\text{batteryChargeUpdate}, charge \leq 30\%, \text{offload}(fid))\}$$
$$\mathcal{P}_l = \{(\text{wifiConnectionReport}, quality < 4, \text{back}(fid))\}$$
$$\mathcal{P}_r = \{(\text{serviceQualityReport}, quality < 7, \text{go}(fid))\}$$

with *fid* being the identifier of f_R. On-site policies $\mathcal{P}_o$ monitor the availability of CPU and battery resources and, if necessary, trigger the offloading action to reduce resource consumption. Once offloading is completed, the policy handler is split into a local and a remote handler (executing, respectively, $\mathcal{P}_l$ and $\mathcal{P}_r$). The former (running $\mathcal{P}_l$) monitors the quality of the wireless connection and decides (possibly, by enacting some forecasting) when it is necessary to request the module back from the cloud service because the connection has become unreliable and in order not to loose the computation performed so far. The latter (running $\mathcal{P}_r$) resides on the cloud service NAM and monitors the quality of the computation service. If not satisfactory (e.g., not sufficiently fast), offloading may become a disadvantage and the module may decide to go to another NAM, possibly its origin one.

Let us now consider the behavior of the remote cloud service, which provides elastic resources to registered users with a positive credit balance. The cloud service provides the users with one or more virtual machines *running a cloned system image*. The mobile device is allowed to offload f_R to a cloned replica for remote execution, thus saving battery and time, since the cloud will speed up the computation. As already mentioned, the offloading process is started on the mobile device by policies of the functional module f_R. On the side of NAMs hosted on virtual machines, policies perform other monitoring tasks such as those described by the following rules:

$$\mathcal{P} = \{(\mathsf{cpuLoadUpdate}, load > 80\%, LoadBalance),\\ (\mathsf{accountCreditReport\_}fid, credit = 0, \mathsf{go}(fid))\}$$

where CPU load is monitored and, if too high, a re-balancing action is executed, moving a functional module to another virtual machine. Furthermore, for each hosted functional module *fid*, the user credit is monitored and, if insufficient, the module is sent back to the owner.

Regarding the policies of the ChordFunctionalModule f_C, among its on-site policies, the following ones regulate its replication on other NAMs:

$$(\mathsf{chordWorkloadUpdate}, workload > 70\%, \mathsf{copy}(fid')),\\ (\mathsf{chordReq}, true, \mathsf{copy}(fid'))$$

with *fid'* being the identifier of f_C. The former policy monitors the workload of the Chord module; if it is overloaded, a copy action is executed to activate a new Chord network node in another NAM. The latter policy performs the same action when a copy of the Chord module is explicitly requested by another NAM that desires to be part of the Chord network.

Figure 4 illustrates a possible interaction, among those allowed by the previously described policies. In particular, nam_1 is hosted on a mobile device and either on a cpuLoadUpdate event or on a batteryChargeUpdate event the policies request an offload action of module f_R.

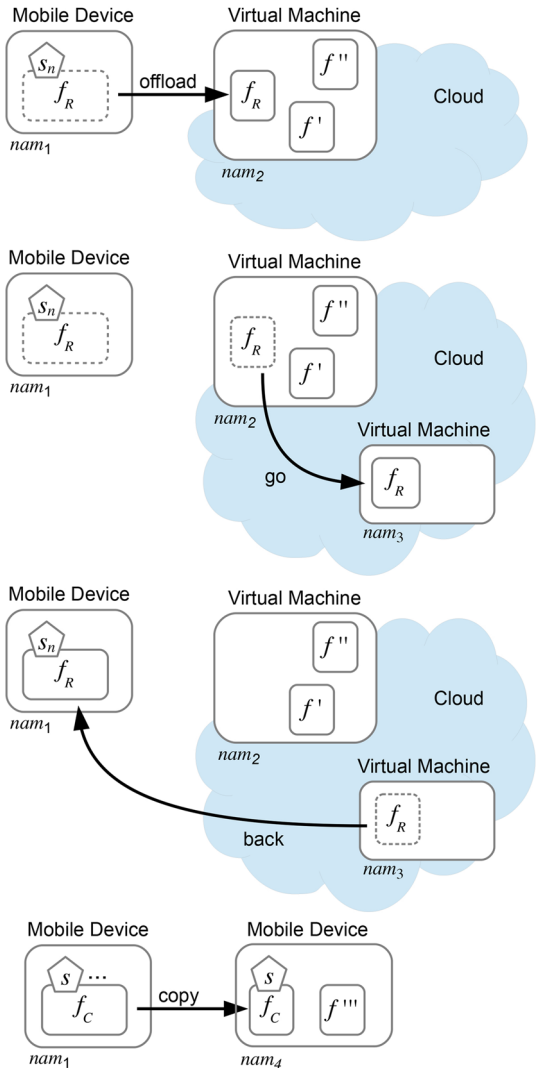

Fig. 4 Possible evolutions of the scenario described in the illustrative example

Therefore, the virtual machine hosting nam_2 accepts module f_R (it may be running other modules). When offloading is complete, all requests of service s_n on nam_1 are redirected to nam_2 for evaluation.

We assume that the virtual machine hosting nam_2 becomes overloaded, which triggers the action moving a functional module to another NAM. This may cause in the underlying cloud middleware the creation of a new virtual machine, but these details are out of the scope of the NAM framework. In Fig. 4, this balancing operation is illustrated as a go action moving f_R to nam_3.

It may be the case that the user moves and the wireless connection becomes weaker and unreliable. This is detected by the local policy handler, as discussed previously. Thus, f_R is requested back by the owner nam_1, so that the execution can continue locally. Also, this event is illustrated in Fig. 4,

Table 2 KLAIM syntax

(Nets)
$N ::= s ::_\rho C \mid N_1 \parallel N_2 \mid (\nu s)N$
(Components)
$C ::= P \mid \langle t \rangle \mid C_1 \mid C_2$
(Processes)
$P ::= a \mid X \mid A(p_1, \ldots, p_n)$
$\mid P_1 ; P_2 \mid P_1 \mid P_2 \mid P_1 + P_2$
$\mid$ **if** (e) **then** $\{P_1\}$ **else** $\{P_2\}$
$\mid$ **while** (e) $\{P\}$
(Actions)
$a ::= \mathbf{in}(T)@\ell \mid \mathbf{read}(T)@\ell \mid \mathbf{out}(t)@\ell$
$\mid \mathbf{inp}(T)@\ell \mid \mathbf{readp}(T)@\ell \mid \mathbf{eval}(P)@\ell$
$\mid \mathbf{newloc}(s) \mid x := e$
(Tuples)
$t ::= e \mid \ell \mid P \mid t_1, t_2$
(Templates)
$T ::= e \mid \ell \mid ?x \mid ?l \mid ?X \mid T_1, T_2$

by showing that f_R goes from nam_3 back to nam_1, on the mobile device.

Finally, the last situation described in Fig. 4 corresponds to the copy of the ChordFunctionalModule f_C to nam_4, running on another mobile device, due to, e.g., an explicit request of the latter NAM.

4 KLAIM

In this section, we summarize the key features of the formal language KLAIM. It has been specifically designed to provide programmers with primitives for handling 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 [19] is a coordination paradigm rather than a language, since it only provides a set of coordination primitives. It relies on the so-called generative communication paradigm, which decouples the 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 retrieving mechanism uses pattern-matching to find the required data in the tuple space.

KLAIM enriches Linda primitives with explicit information about the locality where processes and tuples are allocated. KLAIM syntax[1] is shown in Table 2.

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 a net of the form $N_1 \parallel (\nu s)N_2$, the effect of the operator is to make s invisible from within N_1.

Nodes $s ::_\rho C$ have a unique *locality name* s (i.e., their network address) and an allocation environment ρ, and host a set of components C. The *allocation environment* provides a name resolution mechanism by mapping *locality variables* l (i.e., aliases for addresses), occurring in the processes hosted in the corresponding node, into localities s. The distinguished locality 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 $\langle t \rangle$, composed by means of the parallel operator $C_1 \mid C_2$.

Processes P are the KLAIM active computational units, which can be executed concurrently either at the same locality or at different localities. 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 $P_1 \mid P_2$, 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 their migration to make migrating processes complete) and assume that they are available at any locality of a net. Notably, e ranges over *expressions*, which contain basic values (booleans, integers, strings, floats, etc.) and value variables x, and are formed by using the standard operators on basic values and the non-blocking retrieval actions **inp** and **readp** (explained below). In the rest of this section, we will use the notation ℓ to range over locality names s and locality variables l.

During their execution, processes perform some basic *actions*. Actions $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ are retrieval actions and permit to withdraw/read data tuples from the tuple space hosted at the (possibly remote) locality ℓ: if a matching tuple is found, one is non-deterministically chosen, otherwise the process is blocked. They exploit templates as patterns to select tuples in shared tuple spaces. *Templates* are sequences of actual and formal fields, where the latter are written $?x$, $?l$ or $?X$ and are used to bind variables to values, locality names or processes, respectively. Actions $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ are non-blocking versions of the retrieval actions; namely, during their execution, processes are never blocked. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read** and additionally return the value

[1] We use 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. All such constructs are directly supported by KLAIM related tools (such as, e.g., the analysis tool SAM [26]).

true; otherwise, they return the value *false* and the executing process does not block. **inp**$(T)@\ell$ and **readp**$(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action **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 **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 the other actions, these latter two actions are not indexed with an address because they always act locally.

We conclude the section with a simple example aiming at clarifying how the communication between two KLAIM nodes takes place. Let us consider the following KLAIM net:

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

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

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

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

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

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

5 KLAIM-based semantics for NAM

This section discusses how, from an operational point of view, a NAM network can be defined in terms of a KLAIM net. In particular, the aim of providing the semantics of the NAM framework in terms of the KLAIM formal language is to clarify the relationship among functional modules, their 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., **in**(T) in place of **in**$(T)@$**self**.

A NAM network consisting of a collection of NAMs $\{nam_1, \ldots, nam_m\}$ can be rendered in KLAIM as the following net:

$$nid_1 ::_{\rho_1} (C^1_{TS} \mid C^1_P) \parallel \ldots \parallel nid_m ::_{\rho_m} (C^m_{TS} \mid C^m_P)$$

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

The tuples stored in C_{TS} represent data local to functional modules in $\mathcal{F}$, availability of resources in $\mathcal{R}$, messages to denote service requests or events, code of functional modules in $\mathcal{F}$, and commands to instrument the forms of mobility supported by the framework. We adopt the following convention about tuples: 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.

The processes in C_P, performing the computational tasks and the self-management of the NAM, are defined as the following parallel composition:

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

where:

- *Disp* is a *dispatcher* of service requests to the appropriate functional modules;
- *PMH* is the *policy and mobility handler* that is in charge of enforcing the NAM policies $\mathcal{P}$ and executing the mobility commands;
- F_j includes the processes modeling the functional module f_j in $\mathcal{F}$ with identifier *fid*, i.e., the service handler (SH) and the policy handler (PH) of the functional module, and a number of threads (T), each of which serving a specific service request:

$$SH_{fid} \mid PH_{fid} \mid T^1_{fid} \mid \ldots \mid T^h_{fid}$$

In the rest of this section, we provide some details on the processes mentioned above.

5.1 NAM control

The process that models the service request dispatcher of a NAM is defined as follows:

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

This process cyclically reads (and consumes) a service request, determines the NAM hosting the functional module implementing the service (which can be either the NAM enclosing the dispatcher itself or a remote NAM), and sends a

service assignment to such a NAM. More specifically, a *service binder* tuple of the form $\langle \mathsf{srvBinder}, sid, fid, nid_{IMP} \rangle$, stored in the considered NAM, is used to identify (via pattern-matching) the NAM nid_{IMP} providing the implementation of module *fid* exposing service sid. Depending on whether nid_{IMP} is the local NAM or not, either a local service assignment (tagged by srvAssign) or a remote one (tagged by remoteSrvAssign) is generated.

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

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

Mobility commands are dealt with by the *mobility handler* (*MH*, illustrated in Sect. 5.3), while policies by the *policy handler*. The latter is rendered as a choice composition of the processes modeling Event-Condition-Action rules of the NAM policies $\mathcal{P}$. In particular, an event ev (retrieved by an **in**) triggers the execution of the process P_{act}, realizing the action act, provided that condition co is satisfied.

Note that the *PMH* component enacts mobility actions or policies in a mutually exclusive way. This means that the policy and mobility handler processes only one event at the time, in order to avoid interferences among the executions of different mobility actions, and with the evaluation of policies.

5.2 Functional module control

Every functional module f has a service handler SH_{fid} that has two roles: (1) to react to service assignments, by creating a thread that serves the corresponding service request, and (2) to change state accordingly to mobility requests.

The following KLAIM code models these behaviors:

$$\begin{aligned}
&SH_{fid} = \\
&\quad \mathbf{in}(\mathsf{srvAssign}, ?sid, fid, ?data, ?nid_{SRC}); \\
&\qquad START_THREAD(sid, fid, data, nid_{SRC}); \\
&\qquad SH_{fid} \\
&\quad + \mathbf{in}(\mathsf{copySH}, fid, ?nid_{DST}); \mathbf{eval}(SH_{fid})@nid_{DST}; \\
&\qquad SH_{fid} \\
&\quad + \mathbf{in}(\mathsf{migrateSH}, fid, ?nid_{DST}); \mathbf{eval}(SH_{fid})@nid_{DST} \\
&\quad + \mathbf{in}(\mathsf{offloadSH}, fid, ?nid_{DST}); \mathbf{eval}(RSH_{fid})@nid_{DST}; \\
&\qquad LSH_{fid}
\end{aligned}$$

On arrival of a service assignment (srvAssign) for *fid*, the service handler creates a thread with the following parameters: the service identifier sid, the module identifier *fid*, the $data$ for the computation, and the client identifier nid_{SRC}. We discuss code for thread creation later on. 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, except that it stops its execution. An offload request (offloadSH) behaves differently: It first starts a *remote* service handler RSH_{fid} at location nid_{DST} and then switches to execute a *local* service handler LSH_{fid}. We now introduce the code of these two processes:

$$\begin{aligned}
&LSH_{fid} = \\
&\quad \mathbf{in}(\mathsf{backSH}, fid, ?nid_{DST}); \\
&\quad \mathbf{out}(\mathsf{remoteBackSH}, fid)@nid_{DST}; \\
&\quad SH_{fid} \\
&\\
&RSH_{fid} = \\
&\quad \mathbf{in}(\mathsf{remoteSrvAssign}, ?sid, fid, ?data, ?nid_{SRC}); \\
&\qquad START_THREAD(sid, fid, data, nid_{SRC}); \\
&\qquad RSH_{fid} \\
&\quad + \mathbf{in}(\mathsf{remoteBackSH}, fid) \\
&\quad + \mathbf{in}(\mathsf{goSH}, fid, ?nid_{DST}); \mathbf{eval}(RSH_{fid})@nid_{DST}
\end{aligned}$$

After offloading, service requests are forwarded to the remote NAM nid_{DST} by the dispatcher process *Disp*. Therefore, the solely role of LSH_{fid} is to react to a back request (backSH) by informing the remote NAM (by a remoteBackSH request) and returning to (normal) state SH_{fid}. On the other side, the remote service handler RSH_{fid} has three possible behaviors. The first reacts to a (forwarded) remote service assignment (remoteSrvAssign), by creating a thread to serve the request, and returns to its initial state. The second receives a (forwarded) 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 to location nid_{DST} and terminating. Notice that this case 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}. This check is performed by the process triggering the goSH action, which is the mobility handler *MH* that we will discuss in Sect. 5.3. Clearly, after a go action, service requests are forwarded to the new NAM, where the remote service handler is active. This redirection requires the update of the service bindings, which is again taken care of by the mobility handler process.

Before discussing mobility actions in further detail in the next section, we briefly illustrate how threads are created:

$$\begin{aligned}
&START_THREAD(sid, fid, data, nid_{SRC}) = \\
&\quad \mathbf{read}(\mathsf{srvImpl}, sid, fid, ?Code); \\
&\quad tid := getFreshId(); \\
&\quad \mathbf{out}(\mathsf{thread}, fid, tid); \\
&\quad \mathbf{eval}(Code(tid, data, nid_{SRC}, fid))
\end{aligned}$$

By using the service identifier sid, the implementation ($Code$) of that service in the functional module *fid* is retrieved from a tuple tagged srvImpl. Then, a new thread identifier tid is created and registered as a thread of *fid*. Finally, the thread $Code(tid, data, nid_{SRC}, fid)$ is executed locally. The thread registration phase (with its unique id) is required to be able to retrieve and move running threads of a functional

module when offload/migration of this module is performed. We expect the thread 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 on task completion and to react on migration/offloading requests. We assume the thread user code is instrumented accordingly (and, possibly, automatically).

The policy handler PH_{fid} executes policies similarly to PMH, by using triples (ev, co, act) in the on-site policy $\mathcal{P}_o$ of fid. Furthermore, it reacts to mobility actions identified by tuples with tag in {backPH,copyPH, goPH,migratePH, offloadPH}. In particular, similar to the service handler, in the case of an offload request, it first starts a remote policy handler RPH_{fid} (which executes the functional module remote policy $\mathcal{P}_r$) and then switches to execute a local policy handler LPH_{fid} (which executes the functional module local policy $\mathcal{P}_l$).
To ease the reading, we relegate the code of processes PH_{fid}, RPH_{fid}, and LPH_{fid} to the Appendix.

5.3 Mobility handler

The mobility handler MH executes in mutual exclusion with NAM policies. It is structured as follows:

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

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. We now illustrate the KLAIM code for the offload action handler, then we briefly describe how the other actions are handled. The interested reader can find the corresponding KLAIM code in the Appendix.

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

On arrival of an offload request (offloadReq[2]), the handler first informs the remote NAM nid_{DST} that its NAM (**self**) is the owner (offloader) of the functional module fid, by adding a tuple tagged by offloaderNAM. Then, it *updates* the binder for each service in fid with the new information that the module is at location nid_{DST}. Afterward, it *moves* the implementation of each service (that is, 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 is then expected to react to this mobility request accordingly. Finally, offload requests are sent (locally) to the service handler and to the policy handler by using offloadSH and offloadPH requests, respectively, indicating the destination NAM nid_{DST}. We have seen in the previous section how SH_{fid} reacts to these requests. Finally, the control returns to PMH where either a policy or a mobility request is handled.

The copy action handler CH (whose code is given in the Appendix), on arrival of a copy request (copyReq), performs three operations: (1) it *copies* all binders by setting the remote NAM as the fid location, (2) it *copies* the implementations, and (3) it sends copy requests to the service and policy handler by using copySH and copyPH.

The migrate action handler MiH (whose code is given in the Appendix), on arrival of a migrate request (migrateReq), performs five actions: (1) it *moves* all binders by setting the remote NAM as the fid location, (2) it *moves* the implementations, (3) it *moves* the service assignments, (4) it *moves* the threads, and (5) it sends migrate requests to the service and policy handler by using migrateSH and migratePH.

The back action handler BH (whose code is given in the Appendix), on arrival of a back request (backReq), simply sends to the remote NAM a go request with destination **self**.

Finally, the go action handler GH (whose code is given in the Appendix) has two possible behaviors. The first is performed on the remote NAM, on arrival of a go request (goReq): (1) it retrieves the identity of the local NAM (using a tuple tagged by offloaderNAM), (2) sends a notification (goNotification) to the local NAM with the new location (NAM_2) so that it can update service bindings accordingly, (3) *moves* implementation and threads to the new location, and (4) if the new destination NAM_2 is the originator of the offload, then it is indeed a back action, and it simply sends back requests to the service and policy handler and translates remote service assignments to local ones. Otherwise, it performs three sub-steps: (4.i) it informs NAM_2 of the offloader identity using the offloaderNAM tuple, (4.ii) it sends go requests to the service and policy handler by using goSH and goPH, and (4.iii) it moves remote (not yet served) service assignments. The second behavior of GH is performed on the local NAM and reacts to goNotification messages by updating service binders to point to the new NAM (possibly, **self**).

[2] It is worth noticing that an offload request as received by the offload action handler contains more information (in particular, the identity of the destination NAM) with respect to the corresponding actions specified in the policies (see Sect. 3). We assume indeed that this level of transparency is properly managed by the processes modeling the execution of policy actions (i.e., processes P_{act} within PMH).

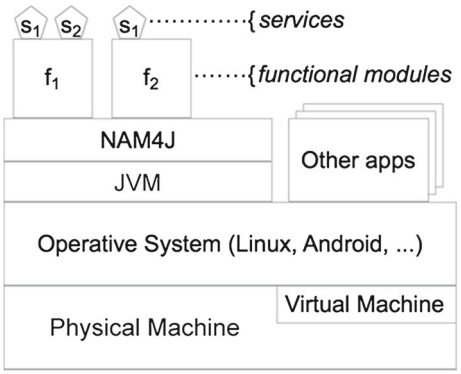

Fig. 5 NAM4J layer stack

6 Formalization at work on NAM4J

NAM4J is a Java middleware which has been specifically developed to implement NAM-based autonomic systems. A layer stack showing the role of NAM4J in a networked system is depicted in Fig. 5. Basically, NAM4J runs in a Java Virtual Machine.

Developers willing to build NAM4J-based applications are required to perform a few steps, reported in the following. The first step is the extension of the NetworkedAutonomicMachine class. Each NAM node has an instance of such a class, which provides data structures and methods to manage all the general activities that characterize the NAM (e.g., loading, specific functional modules). The next step is the definition of the functional modules 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. Finally, developers define the exposed services through the extension of the Service class. As for the functional modules, which are linked to the class extending NetworkedAutonomicMachine, services have to be linked to functional modules through the addConsumableService(Service) and addProvidedService(Service) methods provided by the FunctionalModule class. As described in Sect. 2, each node can only access its own resources, while the interaction with resources of other nodes happens through the respective services.

For example, a NAM interfaced to a number of sensors, measuring different physical quantities, may expose a service for each of them. The services providing sensor data related to a given 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 them through the respective services.

In rest of the section, we illustrate how we have updated NAM4J, in accordance with the Klaim-based semantics for the NAM framework illustrated in Sect. 5. Moreover, we revisit our illustrative example, to show how it has been implemented by means of the updated NAM4J middleware. The source code of the middleware and the example implementation can be browsed on the NAM4J project's web site.[3]

Prior to the re-engineering, the middleware only included two methods to, respectively, request and send code as *jar* or *dex* files. The receiving NAM node subsequently added the included classes to the *classpath* at runtime and began the code execution. Such a mechanism represented a basic implementation of the copy mobility action. During the re-engineering, we developed several new classes grouped into the *mobility* package. Such classes represent the handler, the server-side implementation, and the client-side manager for every mobility action, plus the server-side mobility engine and actions manager. Specifically, the ServerMobilityActionManager class implements the server-side management of the mobility actions and instantiates the proper ActionImplementation object which manages the identification of the code to be migrated and its offloading to the client. The ActionManager class for a specific action implements the client-side behavior by managing the code reception, its runtime adding to the classpath, and its subsequent execution. More details regarding the new classes are given in the following.

Concerning service interaction, we have implemented the dispatcher described in Sect. 5. Specifically, we have defined the IDispatcher interface, which must be implemented by classes characterized by different dispatching algorithms (according to the Strategy pattern [18]). We have also implemented a basic 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.

Figure 6 illustrates how service requests are managed by the basic Dispatcher provided by NAM4J. It is assumed that a discovery module is able to find the NAMs that are able to serve a request—either the local one, or remote ones. The discovery module selects a NAM among those offering a functional module capable of dealing with the service request and inserts the request in the queue associated with its dispatcher. Then, there are two different cases of service request assignments. In the first case (Fig. 6, 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 execute() on the destination functional module, passing the description of the requested service. The destination functional module has a threadpool, to handle incoming execution requests. Once a requested execution completes, a reply is returned to the

[3] https://code.google.com/p/nam4j/source/browse/.

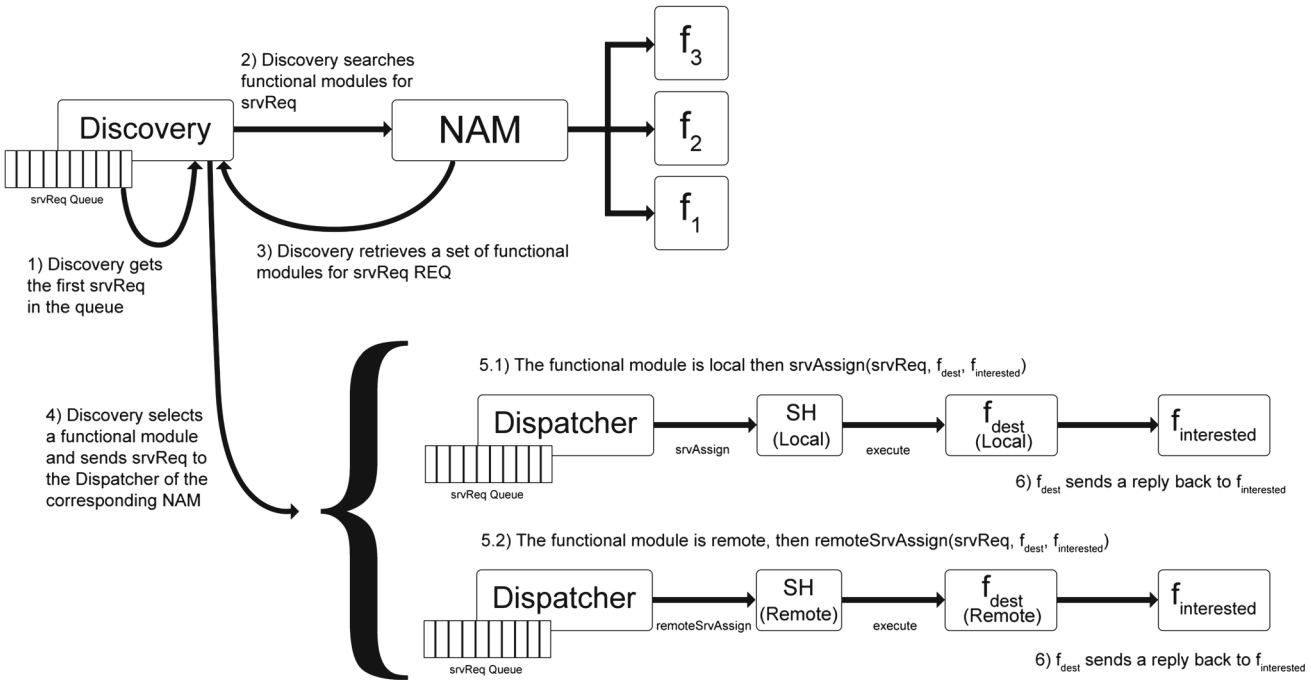

Fig. 6 Operations of the basic Dispatcher provided by NAM4J

interested functional module, i.e., the functional module that initially sent the service request to the Dispatcher. In the second case (Fig. 6, 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 then calls execute() on the destination functional module, passing the description of the requested service. Once the requested execution completes, a reply is directly returned to the interested functional module. Notably, discovery functionalities and details concerning services provision considered here are out of the scope in the formalization, which mainly focusses on mobility actions semantics.

Then, we have implemented an IMobilityEngine interface and a basic MobilityEngine class which manages the offloading, copy, or migration of a functional module. As illustrated in Fig. 7, depending on the case, the MobilityEngine interacts with the most suitable MobilityHandler (in this case, the State Pattern is used [18]). Let us focus on the CopyActionHandler. It is provided with a queue, to handle incoming copy requests. Once a request exits the queue, the CopyActionHandler performs the following actions:

1. tells the remote NAM that a functional module f is going to be copied;
2. tells the ServiceHandler associated with f that f is going to be copied to the remote NAM, in order to create also a copy of the ServiceHandler;
3. sends the binaries of f and the copy of the ServiceHandler to the remote NAM;
4. activates f and its ServiceHandler in the remote NAM.

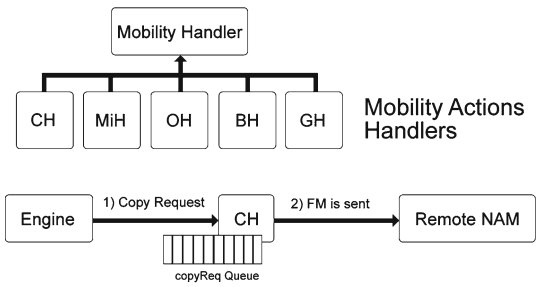

Fig. 7 Operations of the basic MobilityHandler provided by NAM4J and details of the copy action

6.1 Implementation of the illustrative example

This section describes an example in which two NAM nodes interact through the copy mobility action. To better explain the steps performed during the implementation, several code examples are provided. Thus, the section represents a brief tutorial on how to implement NAM-based applications, using NAM4J.

With reference to the code available online, package it.unipr.ce.dsg.examples.migration includes the TestCopyAction class, which represents a demo for the copy mobility action, based on the example described in Sect. 3. NAM n_1 acts as a client and requests a module ChordFunctionalModule to a known NAM n_2, which acts as a server. Such a functional module will allow n_1 to enter the Chord network and start interacting with the other nodes. Moreover, for illustration purposes, as presented by Listing 6.1,

n_1 requires a generic functional module and a generic service, which are described by the TestFunctionalModule and TestService classes of the same package. The request for ChordFunctionalModule has two null values as second and third parameters, since no service is required to be copied,[4] while the second request has the name and the identifier of the required service. Such parameters are String arrays, as they specify the list of names and identifiers for all the required services.

caption

```
1  /* Request Chord functional module */
2  migration.startCopyAction("
       ChordFunctionalModule",
3                        null,
4                        null,
5                        Platform.DESKTOP);
6
7  /* Request a functional module and a Service
       */
8  migration.startCopyAction("
       TestFunctionalModule",
9                        new String[] { "
       TestService" },
10                       new String[] { "
       serviceId" },
11                       Platform.DESKTOP);
```

The it.unipr.ce.dsg.nam4.impl.mobility package contains five abstract classes, which provide the base structure for the server-side mobility action handlers. Such classes implement the Runnable interface, so that the management of each action happens in a separate thread. The package also includes five classes, which extend the aforementioned abstract classes to provide a default implementation, and the ServerMobilityActionManager class, which accepts the action requests and instantiates the suitable implementation class.

Currently, just the copy action has been implemented by the CopyActionImplementation and ClientCopyActionManager classes, which provide server-side and client-side management methods, respectively.

To request the execution of a copy action, the NetworkedAutonomicMachine class provides the startCopyAction method reported in Listing 6.1. Such a method takes as parameters the name of the main class of the functional module to be copied (the corresponding *jar* will be found by inspecting the local libraries, using reflection), optional service names and ids to be copied along with the functional module, and the platform on which the client is running—either ANDROID or DESKTOP. The server uses the first parameter to scan its own *jar* or *dex* files and find the one to be sent. The last parameter is mandatory, because the Android *Dalvik Virtual Machine* requires *dex* archives instead of the more common *jar*s. Listing 6.1 presents the code that instantiates ClientCopyActionManager and delegates its execution to a thread of a pool. The Action.COPY parameter is a string identifying the required mobility action.

caption

```
1  public void startCopyAction(
2      String functionalModule,
3      String[] service,
4      String[] serviceId,
5      Platform clientType) {
6          ClientCopyActionManager ccam = new
       ClientCopyActionManager(this,
7                      functionalModule,
8                      service,
9                      serviceId,
10                     clientType,
11                     Action.COPY);
12         poolForClientMobilityAction.execute(
       ccam);
13     }
```

As shown by Listing 6.1, ClientCopyActionManager's run method, which is executing on a NAM acting as a client, calls the findRemoteItem method, which opens a socket connection to a known NAM acting as a server. The client then sends over the socket the Action.COPY string, its platform type, and the name of a class that identifies the required functional module, as shown by Listing 6.1. If the user asks for one or more services, the method repeats the request sequence for each of them—see the for cycle at line 12, where requiredServiceClass is the array of required services.

caption

```
1  FunctionalModule fm = (FunctionalModule)
       findRemoteItem(
2      requiredFmClass,
3                                  clientType,
4      MigrationSubject.FunctionalModule,
5                                  action);
6
7  nam.addFunctionalModule(fm);
8
9  /* Check if the client asked for one or more
       services of the functional module to
       get copied */
10 if (requiredServiceClass != null) {
11
12 for (int g = 0; g < requiredServiceClass.
       length; g++) {
13
14     String currentServiceClassName =
       requiredServiceClass[g];
15     String currentServiceId =
       requiredServiceId[g];
16
17     if (currentServiceClassName != null &&
       currentServiceId != null) {
18
19         /* Obtaining the Service */
20         Service serv = (Service)
       findRemoteItem(
21                 currentServiceClassName,
22                 clientType,
```

[4] For the sake of simplicity, the formalization does not consider the possibility of copying a functional module without the associated services. However, we do not envisage any difficulties on modeling this feature.

```
            MigrationSubject.SERVICE,
            action);

            /* Adding the Service to the
    functional module */
            fm.addProvidedService(
    currentServiceId, serv);
        }
    }
}
```

Listing 6.1 presents the code that allows for server-side management of incoming requests, by passing each one to a ServerMobilityActionManager instance.

caption

```
Socket cs = null;
ServerSocket ss = null;
ss = new ServerSocket(serverPort);

while (true) {
    cs = ss.accept();
    poolForServerMobilityAction.execute(new
    ServerMobilityActionManager(cs, this));
}
```

As previously stated, ServerMobilityActionManager's run method receives the requested action name and instantiates the corresponding management class, as illustrated by Listing 6.1.

caption

```
switch (action) {
    case COPY: {

        copyActionImplementation = new
    CopyActionImplementation(this.nam, is,
    os);

        Thread copyActionThreadStart = new
    Thread(copyActionImplementation);

        copyActionThreadStart.start();

        break;
    }
    /* ... */
}
```

Finally, CopyActionImplementation's run method identifies the *dex* or the *jar* file by searching for the requested class name inside all the Java archives owned by the node, and subsequently performs the main activity of the copy action, i.e., it sends on the socket a description of the functional module, or service, and the file itself.

7 Related work

This section discusses related work in the fields of MCC, autonomic middleware, code migration, and their formalization.

7.1 Mobile cloud computing

Many approaches to MCC have been proposed in the literature. In [24], three reference MCC approaches are identified. They 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. The results from the augmented execution are reintegrated upon completion. *Elastically Partitioned Applications* can improve their performance by delegating part of the application to remote execution on a resource-rich cloud infrastructure. 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. In a recent work [21], we have proposed a fourth approach, called *Hybrid Mobile Cloud*, which consists of two or more mobile devices that 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.

In this work, we have considered an Augmented Execution case study to illustrate our formalization. In the near feature we plan to extend our study to the other three approaches.

For further details on the motivations underlying the MCC paradigm, concerning, e.g., energy saving and increase of performance, we refer the interested reader to the seminal work [25] on offloading. For surveys on the current state-of-the-art on MCC approaches and technologies, we refer to [15,16,24].

7.2 Autonomic middleware

Autonomic Computing brings together many sub-fields of computer science, with the purpose of creating computing systems that manage themselves. Autonomic principles should also drive MCC systems, where mobile devices have to monitor themselves and take (or not) offloading decisions. A widely known model for autonomic control loops is MAPE-K [22], characterized by the following steps: Monitor (by means of sensors), Analyze, Plan, and Execute (by means of effectors), using a shared base of Knowledge. Among available MAPE-K implementations, the Autonomic Computing Toolkit is a collection of self-managing autonomic technologies [28]. Also, the ABLE Toolkit [6] offers autonomic management in the form of a multi-agent architec-

ture, in which the *autonomic manager* is an agent or a set of agents. Kinesthetics eXtreme [23,32] is an implementation of the MAPE-K loop, whose main purpose is the addition of autonomic properties to legacy systems.

In our work, the main ingredient we borrow from the Autonomic Computing domain is the use of policies for monitoring the system execution and its working environment, and hence performing the appropriate adaptation actions when needed. Typically, these policies are expressed as Event-Condition-Action (ECA) rules. The ECA paradigm was originally introduced for active databases [11] and then applied to the design of policy languages. Although recently more sophisticated forms of policies have been considered for supporting Autonomic Computing, as, e.g., in [14], we preferred here to rely on ECA rules to keep this aspect of our formal development clearer, but still effective.

With respect to MAPE-K and derived approaches, the NAM framework we consider (and improve) in this work is somehow less constrained—for example, the use of a knowledge base is not mandatory. The NAM approach is directed to the development of distributed middleware—conversely, networks are not considered in MAPE-K. Like MAPE-K, NAM is not tight to a specific implementation. NAM4J is the current Java-based incarnation, but nothing prevents one from developing other NAM-based software tools.

7.3 Code migration

Code mobility is the capability to dynamically reconfigure, at runtime, the bindings between the software components of the application and their physical location within a computer network [9]. Two possible scenarios exist: (1) *strong mobility*, if units are allowed to move their code and execution state to a different location and (2) *weak mobility*, if a unit executing in a certain location is allowed to dynamically bind to code coming from a different site (i.e., the execution state is not moved). In Java, migrating the code segment and the data space of a thread is feasible, while relocation of the execution state of a thread to another Java Virtual Machine (JVM) is still debated in the mobile code community. Strong mobility support has been provided in [8] to server applications, by extending the scheduler of the IBM Jikes Research Virtual Machine (RVM). Unfortunately, this approach cannot be applied to mobile platforms. Regarding Android, for example, the Dalvik Virtual Machine cannot be replaced by the Jikes RVM. Other researchers chose to deal with Java strong mobility from the inside, by modifying the bytecode interpreter to keep track of the execution state [7,34]. Neither this approach can be applied to applications running on mobile devices. On the iOS platform, strong mobility is unfeasible, due to the SDK constraints imposed by Apple. Coming from our previous experience with the SP2A middleware [2], NAM4J currently supports only weak mobility, while strong mobility is work in progress. Anyway, in our NAM formalization, we have already considered both forms of mobility.

7.4 Mobile and autonomic computing formalizations

In the literature, many linguistic formalisms for modeling different forms of mobility are proposed. Most of them are based on π-calculus [29], which in its standard definition directly allows only the mobility of links between linked processes (process mobility is enabled in the higher-order variant of the calculus). Some of such formalisms, namely KLAIM, Dπ, Djoin, and Ambient, are surveyed and compared in [17]. Among them we have selected KLAIM as basis for our formalization because, besides (strong and weak) mobility mechanisms, it provides a natural way to model the architecture of NAM-based systems. In particular, a NAM network is rendered as a network of KLAIM nodes. Each node is then equipped with a tuple space modeling data local to functional modules, availability of resources, messages, code of functional modules, and mobility commands. The processes running on nodes represent both service threads and NAM management components (i.e., the dispatcher, the policy and mobility handler, and the functional modules).

Regarding autonomic computing, most of the proposals in the literature still concern full-fledged programming languages rather than foundational models. Some proposed formalisms, as in [4,5,35], are inspired by chemical and biological phenomena. A formalism closer to programming languages, following a process calculi approach and based on KLAIM, is SCEL [14]. Although SCEL is equipped with constructs for dealing with autonomicity, it mainly provides communication primitives for dealing with ensembles that are not relevant for our study and make the operational semantics much more complex. In more practical terms, SCEL is not currently equipped with verification tools, which we plan to use to analyze MCC-based applications. Instead, KLAIM has the advantage of conveniently enabling the modeling of autonomic features (as shown in [20]) and, moreover, of coming with software tools that support various forms of analysis.

A combination of both mobility and autonomicity is necessary for proper modeling of MCC scenarios. Therefore, KLAIM turns out to be a natural choice for this task.

8 Conclusions

We have formalized a framework and some key primitives to support the design of MCC systems. Specifically, we have adopted NAM as a conceptual model for MCC and KLAIM as a formalization language. In particular, we have clarified the role of policies as means to enact autonomic and context-aware mobility strategies. Moreover, we have shown our for-

mal approach at work on an illustrative example, including not only offloading but also other cost- and reliability-driven strategies.

Regarding future work, we envisage the following research lines. First, we plan to apply existing analysis tools for verifying MCC systems specified at high level of abstraction. The choice of KLAIM has the advantage of supporting this task by means of the SAM tool [26]. The challenge here is the identification of relevant and desirable properties for MCC.

We also intend to thoroughly analyze issues about costs of data transmission, which may affect the decision whether to offload or not a computation to another NAM. Currently, they can be dealt with using policies. Indeed, policies act as evaluation points that can take into account cost issues. For example, a policy can include some conditions involving data transmission aspects (e.g., $WiFiConnectionQuality < 4$, $FunctionModuleSize > 15$MB, etc.). However, the evaluation of a policy is based on the current status of the system and of its environment. A more sophisticated decision support could be defined by exploiting the model provided by the KLAIM-based semantics. In particular, the stochastic extension of KLAIM [13], accepted as input by SAM, permits enriching KLAIM models with stochastic aspects that enable the evaluation (possibly, at runtime) of performance and other quantitative parameters. In this way, cost aspects would be expressed in terms of stochastic parameters of the model, and model checking techniques would be exploited to support effective decision making in mobility strategies, by taking into account also future actions to perform and conditions to meet. NAM4J could be used to extract information from execution traces, to determine the appropriate parameters for the stochastic models.

Another research direction we plan to pursue concerns the extension of the language for expressing policies. Currently, the NAM framework relies in policies defined as Event-Condition-Action rules, and an implementation of the Mobility Engine based on Drools[5] is going to be introduced in NAM4J. Although this is a well-known and largely adopted approach, it has some limitations, mainly concerning policy compositionality. Thus, we will consider languages devised for expressing more structured and sophisticated forms of policies, such as XACML [31] and FACPL [27].

Last but not least, we are interested in the formal specification of stateful services in NAM. Currently, we consider stateless services, i.e., they do not keep track of previous interactions with clients. However, as we stated in Sect. 2, services are just entry points to functional modules. Thus, nothing prevents to have stateful services, if the associated functional modules take into account previous executions—as suggested, for example, by the Web Service Resource Framework (WSRF) [30].

[5] https://www.jboss.org/drools/.

Appendix

In this appendix we report and briefly comment the entire KLAIM specification of the NAM framework formalization. We start by reviewing the various kinds of tuples used to synchronize the NAMs and to realize mobility actions.

Control tuples

Service identifiers are bound to functional modules that can offer those services and may be located in a local or remote NAM; services are implemented within a functional module by a process $Proc$:

$\langle \text{srvBinder}, sid, fid, nid \rangle$ $\langle \text{srvImplem}, sid, fid, Proc \rangle$

Services are accessed through a service request and then dispatched to a specific functional module *fid* by a service assignment, if it is a local module, or by a remote service assignment if it is an offloaded module:

$\langle \text{srvReq}, sid, data, nid \rangle$
$\langle \text{srvAssign}, sid, fid, data, nid \rangle$
$\langle \text{remoteSrvAssign}, sid, fid, data, nid \rangle$

Whenever a functional module is offloaded, the host NAM is aware of the identity nid of the offloading NAM so that it is able to send the module back to the owner on need:

$\langle \text{offloaderNAM}, fid, nid \rangle$

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

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

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

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

and to its policy handler:

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

Running threads are associated with a functional module and have their own unique identifier tid, used when migrating or offloading the module:

$\langle \text{thread}, fid, tid \rangle$

When a migrate/offload action is performed, move requests are issued for each thread:

$\langle \text{moveThread}, tid \rangle$

We expect that thread code is suitably instrumented to handle these move requests and threads behave accordingly.

NAM control

On arrival of a service request, the dispatcher chooses the appropriate functional module to provide the service:

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

The policy and mobility handler runs policies and realizes mobility actions:

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

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

The copy action handler, on arrival of a copy request, *copies* all binders by setting the remote NAM as the *fid* location, *copies* the implementations, and sends copy requests to the service and policy handler:

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

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

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

The offload action handler has been discussed in the paper:

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

The back action handler BH, on arrival of a back request, sends to the remote NAM a go request with destination **self**:

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

The go action handler has two possible behaviors. The first is performed on the remote NAM and, on arrival of a go request, retrieves the identity of the offloader NAM, sends a notification (goNotification) to the offloader NAM with the new location (NAM_2) so that it can update service bindings accordingly, and *moves* implementation and threads to the new location. If the new destination NAM_2 is the offloader itself, then it is indeed a back action, and it 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 performed on the local NAM and reacts to goNotification messages by updating service binders to point to the new NAM (possibly, **self**).

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

Functional module control

The service assignment handler, in its normal mode operation, has been described in the paper:

$SH_{fid} =$
  in(srvAssign, ?sid, fid, ?$data$, ?nid_{SRC});
    $START\_THREAD(sid, fid, data, nid_{SRC})$;
    SH_{fid}
  + **in**(copySH, fid, ?nid_{DST}); **eval**(SH_{fid})@nid_{DST};
    SH_{fid}
  + **in**(migrateSH, fid, ?nid_{DST}); **eval**(SH_{fid})@nid_{DST}
  + **in**(offloadSH, fid, ?nid_{DST}); **eval**(RSH_{fid})@nid_{DST};
    LSH_{fid}

In the paper, we have also discussed the behavior of the service assignment handler in its offloaded mode operation:

$LSH_{fid} =$
  in(backSH, fid, ?nid_{DST});
    out(remoteBackSH, fid, nid_{DST})@nid_{DST};
    SH_{fid}

$RSH_{fid} =$
  in(remoteSrvAssign, ?sid, fid, ?$data$, ?nid_{SRC});
    $START\_THREAD(sid, fid, data, nid_{SRC})$;
    RSH_{fid}
  + **in**(remoteBackSH, fid, _)
  + **in**(goSH, fid, ?nid_{DST}); **eval**(RSH_{fid})@nid_{DST}

Similar to the service assignment handler, the policy handler has a normal mode operation, where policies in P_N are executed on local events and mobility actions are handled in a similar way:

$PH_{fid} =$
  $\sum_{(ev,co,act)\in P_N}$ **in**(event, ev); **if** (co) **then** {P_{act}}; PH_{fid}
  + **in**(copyPH, fid, ?nid_{DST}); **eval**(PH_{fid})@nid_{DST};
    PH_{fid}
  + **in**(migratePH, fid, ?nid_{DST}); **eval**(PH_{fid})@nid_{DST};
  + **in**(offloadPH, fid, ?nid_{DST}); **eval**(RPH_{fid})@nid_{DST};
    LPH_{fid}

In offloaded mode, the policy handler splits into a local and a remote handler, reacting to local and remote events:

$LPH_{fid} =$
  $\sum_{(ev,co,act)\in P_L}$ **in**(event, ev); **if** (co) **then** {P_{act}}; PH_{fid}
  + **in**(backPH, fid, ?nid_{DST});
    out(remoteBackPH, fid, nid_{DST})@nid_{DST};
    PH_{fid}

$RPH_{fid} =$
  $\sum_{(ev,co,act)\in P_R}$ **in**(event, ev); **if** (co) **then** {P_{act}}; PH_{fid}
  + **in**(remoteBackPH, fid, _)
  + **in**(goPH, fid, ?nid_{DST}); **eval**(RPH_{fid})@nid_{DST}

Also in this case, mobility actions are handled similarly to the service handler.

Macros

We now illustrate some macro code that helps improving code readability and performs crucial operations of the mobility handling process. In these macros, we use the **while** construct with the non-blocking variants of **in**/**read** as argument, so we are ensured to consider each tuple of interest at least once. In case of **read** argument, we assume that the semantics of **while** ensures that each tuple is considered at most once. Notably, the **while** loops in our macros code are ensured to terminate, due to a disciplined use of the considered tuples and appropriate boolean conditions on some of their fields.
Service binders can be moved, copied, and updated:

$MOVE\_BINDER(fid, nid_{DST}) =$
  while (**inp**(srvBinder, ?sid, fid, ?nid_{IMP}))
    {**out**(srvBinder, sid, fid, nid_{DST})@nid_{DST}}

$COPY\_BINDER(fid, nid_{DST}) =$
  while (**readp**(srvBinder, ?sid, fid, **self**))
    {**out**(srvBinder, sid, fid, nid_{DST})@nid_{DST}};

$UPDATE\_BINDER(fid, nid_{DST}) =$
  while (**inp**(srvBinder, ?sid, fid, ?nid_{IMP}) && $nid_{IMP} != nid_{DST}$)
    {**out**(srvBinder, sid, fid, nid_{DST})}

In the first case, each binder is deleted locally and written in the remote location, with a pointer to the remote implementation (we move implementations accordingly). In the second case, we do not consume local binders, but we still update the implementation location in the copy. In the third case, we change the implementation location by replacing those binders that still point to the local implementation (we assume nid_{IMP} and nid_{DST} are different).
Service assignments can be moved (in their local and remote variants) and also translated from local to remote and back:

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

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

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

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

Local to remote translation is necessary to move not-yet-served requests in offloading/migration, so that no request is lost. Similarly, remote to local translation is used when offloading is terminated, in a back action.

Implementations can simply be moved or copied:

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

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

Finally, we consider macros to handle threads. These can only be moved or started (forced termination is not allowed):

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

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

Moving threads of a functional module during offloading or migration is performed by retrieving (and deleting) each thread identifier associated with that module, sending a moveThread message (thus relying on the thread ability to react to these requests), and registering the thread in the remote location.

References

1. Amoretti M, Grazioli A, Senni V, Tiezzi F, Zanichelli F (2014) Towards a formal approach to mobile cloud computing. In: Parallel, distributed, and network-based processing (4PAD session), pp 743–750. IEEE
2. Amoretti M, Laghi MC, Tassoni F, Zanichelli F (2010) Service migration within the cloud: code mobility in SP2A. In: High performance computing and simulation (HPCS), 2010 international conference on, pp 196–202
3. Amoretti M, Picone M, Zanichelli F (2012) Global ambient intelligence: an autonomic approach. In: Pervasive computing and communications workshops (PERCOM), pp 842–847. IEEE
4. Andrei O, Kirchner H (2009) A higher-order graph calculus for autonomic computing. In: Graph theory, computational intelligence and thought, volume 5420 of LNCS, pp 15–26. Springer
5. Banâtre JP, Radenac Y, Fradet P (2004) Chemical specification of autonomic systems. In: Intelligent and adaptive systems and software engineering (IASSE), pp 72–79. ISCA
6. Bigus JP, Schlosnagle DA, Pilgrim JR, Mills WN III, Diao Y (2002) ABLE: a toolkit for building multiagent autonomic systems. IBM Syst J 41(3):350–371
7. Bouchenak S, Hagimont D, Krakowiak S, De Palma N, Boyer F (2002) Experiences implementing efficient java thread serialization, mobility and persistence. In: I.N.R.I.A., Research report no. 4662
8. Cabri G, Leonardi L, Quitadamo R (2006) Enabling java mobile computing on the IBM jikes research virtual machine. In: Principles and practice of programming in java (PPPJ), volume 178 of ACM international conference proceeding series, pp 62–71. ACM
9. Carzaniga A, Picco GP, Vigna G (2007) Is code still moving around? Looking back at a decade of code mobility. In: ICSE companion volume, pp 9–20. IEEE Computer Society
10. Da Silva D (2013) Opportunities for autonomic behavior in mobile cloud computing, 2013. Keynote talk at ICAC'13. Available at https://www.usenix.org/conference/icac13/title-tba-0
11. Dayal U, Hanson EN, Widom J (1994) Active database systems. In: Modern database systems, pp 434–456. ACM
12. De Nicola R, Ferrari GL, Pugliese R (1998) KLAIM: a kernel language for agents interaction and mobility. IEEE Trans Softw Eng 24(5):315–330
13. De Nicola R, Katoen JP, Latella D, Loreti M, Massink M (2007) Model checking mobile stochastic logic. Theor Comput Sci 382(1):42–70
14. De Nicola R, Loreti M, Pugliese R, Tiezzi F (2014) A formal approach to autonomic systems programming: the SCEL language. ACM Trans Auton Adapt Syst (To appear)
15. Dinh HT, Lee C, Niyato D, Wang P (2013) A survey of mobile cloud computing: architecture, applications, and approaches. Wirel Commun Mob Comput 13(18):1587–1611
16. Fernando N, Loke SW, Rahayu W (2013) Mobile cloud computing: a survey. Future Gener Comput Syst 29(1):84–106
17. Ferrari GL, Pugliese R, Tuosto E (2000) Calculi for network aware programming. In : Workshop "from objects to agents": evolutive trends of software systems (WOA), pp 23–28. Pitagora Editrice Bologna
18. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing
19. Gelernter D (1985) Generative communication in Linda. ACM Trans Program Lang Syst 7(1):80–112
20. Gjondrekaj E, Loreti M, Pugliese R, Tiezzi F (2012) Modeling adaptation with a tuple-based coordination language. In: Symposium on applied computing (SAC), pp 1522–1527. ACM
21. Grazioli A, Picone M, Zanichelli F, Amoretti M (2013) Code migration in mobile clouds with the NAM4J middleware. In: Mobile data management (MDM), 2013 IEEE 14th international conference on
22. IBM (2005) An architectural blueprint for autonomic computing. Technical report, IBM Corporation, Third edition
23. Kaiser G, Parekh J, Gross P, Valetto G (2003) Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems. In: Active middleware services (AMS), pp 22–30. IEEE Computer Society
24. Kovachev D, Klamma R (2012) Beyond the client-server architectures: a survey of mobile cloud techniques. In: Communications in China workshops (ICCC), pp 20–25. IEEE Computer Society
25. Kumar K, Lu Y-H (2010) Cloud computing for mobile users: can offloading computation save energy? IEEE Comput 43(4):51–56
26. Loreti M (2010) SAM: stochastic analyser for mobility, 2010. Available at http://rap.dsi.unifi.it/SAM/
27. Margheri A, Masi M, Pugliese R, Tiezzi F (2013) A formal software engineering approach to policy-based access control. Technical report, DiSIA, Univ. Firenze, 2013 Available at http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf
28. Melcher B, Mitchell B (2004) Towards an autonomic framework: self-configuring network services and developing autonomic applications. Intel Technol J 8(4):279–290
29. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, I and II. Inf Comput 100(1):1–40, 41–77
30. Oasis WSRF TC (2006) Web services resource framework (WSRF) v1.2, April 2006. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
31. OASIS XACML TC (2013) eXtensible access control markup language (XACML) version 3.0, January 2013. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

32. Parekh J, Kaiser G, Gross P, Valetto G (2003) Retrofitting autonomic capabilities onto legacy systems. Tech. Rep. CUCS-026-03, Columbia University
33. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for Internet applications. In: Applications, technologies, architectures, and protocols for computer communications (SIGCOMM), pp 149–160. ACM
34. Suri N, Bradshaw JM, Breedy MR, Groth PT, Hill GA, Jeffers R, Mitrovich TS (2000) An overview of the NOMADS mobile agent system. In: ECOOP workshop on mobile object systems
35. Viroli M, Pianini D, Montagna S, Stevenson G (2012) Pervasive ecosystems: a coordination model based on semantic chemistry. In: Symposium on applied computing (SAC), pp 295–302. ACM