Towards a Formal Approach to Mobile Cloud Computing

Michele Amoretti
SITEIA.PARMA
Univ. of Parma, Italy
michele.amoretti@unipr.it

Alessandro Grazioli and Francesco Zanichelli
Dep. of Information Engineering
Univ. of Parma, Italy
grazioli@ce.unipr.it, francesco.zanichelli@unipr.it

Valerio Senni and Francesco Tiezzi
IMT Advanced Studies Lucca, Italy
valerio.senni@imtlucca.it
francesco.tiezzi@imtlucca.it

***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 NAM framework is a general tool to describe networks of hardware and software autonomic entities, providing or consuming services or resources, that can be applied 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 restricted set of mobility actions supporting MCC. The second contribution is a formal semantics for those actions, which allows us to better understand the behavior of MCC systems and paves the way for the application of formal reasoning techniques. As an outcome, we also derive a more precise formalization of the core NAM features, which may contribute to further development of that framework and the related middleware.**

I. 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 [18]. 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. Moreover, as multiple offloading approaches are possible [17] depending on the task and context, autonomic computing techniques appear promising to increase the robustness and flexibility of MCC systems [8]. 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 [1] 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, for the purpose of reasoning on MCC aspects, the middleware contains too low-level details while the conceptual framework is too abstract.

The aim of this paper is to provide the NAM framework with a formal base in terms of an *operational semantics*, 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. For this purpose, we use the Kernel Language for Agents Interaction and Mobility (KLAIM) [10], which 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. The formalization process contributed to a clarification and refinement of the NAM framework with specific focus on MCC features. In addition, we have analyzed many typical scenarios arising in MCC applications, from which we have identified and formalized five different mobility primitives that can be employed in high-level design of MCC applications.

This formalization effort provides a common conceptual model towards: *(i)* a better understanding of MCC issues, *(ii)* the verification of relevant properties of MCC systems, and *(iii)* formal-based design of autonomic context-aware decision strategies.

The remainder of this paper is structured as follows. Section II presents a simplified description of what a NAM is, tailored to our formalization purpose. Section III describes NAM at work on a typical case study from the MCC domain. Section IV outlines the main features of KLAIM, which are used in Section V to define a formal semantics of NAM. Section VI describes related work regarding MCC, autonomic middleware, code migration, and their formalization. Finally, Section VII reports our conclusions and describes future work.

II. The NAM framework

A system of Networked Autonomic Machines (NAMs) is a loosely connected network of hardware/software entities, which provide or consume services or resources. In this paper we focus on specific MCC aspects, like data and code mobility, and thus we only consider the set of NAM concepts devoted

to address them. Other NAM concepts, such as resource monitoring, service composition, 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 (both *application data* and *awareness data*, such as sensor readings and context events) and computational entities (service threads exploiting functionalities provided by libraries called *functional modules*). More formally, a NAM is represented as a tuple $nam = \langle nid, R, F, P \rangle$, where nid is the NAM identifier, R is a set of physical *resources*, $F = \{f_1, \ldots, f_m\}$ is a set of functional modules (FMs), and P is a set of (self-management) policies. We do not consider data as a resource and we assume it is always stored within FMs and moved accordingly. More general models including data in NAMs 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 R, representing available resources, and F describing FMs that currently reside on it. Autonomic policies are a crucial means to support MCC, since they alleviate the 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 FM is represented as a tuple $f = \langle fid, S, P_f, D, T \rangle$, where *fid* is the functional module identifier, S is a set of bindings from service names to methods of f implementing them, P_f is a tuple containing functional and self-management policies, D is a set of data available to the module, and T is a set of threads currently run by the module itself. We consider a *service* as an entry point for a FM, which has the role of aggregating functions and data to provide computational tasks. In other words, functions hosted by FMs are accessed by other NAMs or FMs via services. To this purpose, when a FM receives a service request, it identifies (via bindings in 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, FM policies $P_f = \langle P_o, P_l, P_r \rangle$ are structured in three parts: P_o are the *on-site* policies, active when the module is not offloaded, while (P_l, 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 recovery of the module. Similarly, a remote event can arise on lack of resources, triggering the decision of sending the module back to the owner.

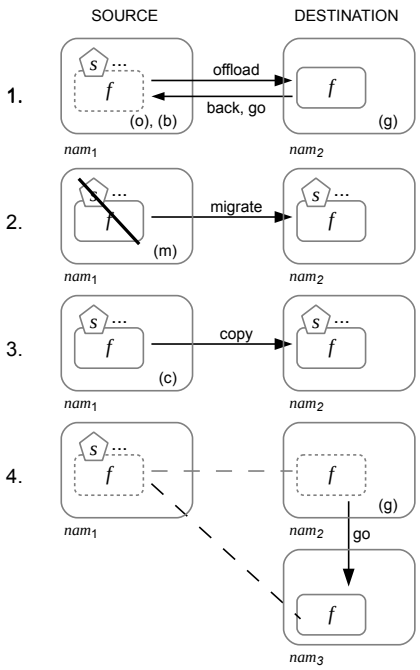

Figure 1. Allowed mobility actions. Capabilities of source/destination NAMs are indicated by the following tags: (o)=offload, (b)=back, (g)=go, (m)=migrate, and (c)=copy

A. Mobility Actions

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

In the first scenario, nam_1 is lacking resources (such as battery or cpu) so it decides, according to its internal policies, to move the code of FM f to nam_2 through an offload action. As an effect of this action, the resource-consuming elements of f (i.e., data D and running threads T) are moved to nam_2 and are regulated by specific policies P_r (while, from now on, policies P_l are activated and enforced locally to nam_1). Therefore, f stops consuming resources of the source and starts consuming the ones of the destination. The entry points of f (i.e., the service specified in S) are, instead, left on nam_1. This choice is motivated by the need of full transparency of offloading with respect to local and remote modules that use services of f. This operation requires 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 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 (e.g. 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 an action migrate. After this action, no part of f (including services) is available on nam_1. Clearly, this action requires to update the set F_1 of functional modules

on nam_1 as well as the set 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 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 offloading any more. Thus it moves f to nam_3 through a go action. This operation moves all elements of f in nam_2 to nam_3 and updates 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 T of the module f is suspended and, then, recovered in the remote location. Similarly, local data D of the module is 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 D of data and T of threads are initially empty in nam_2.

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

III. Case Study

In this section, we show the (fragment of) NAM framework, described in Section II and formalized in Section V, at work in a simple, but realistic, MCC case study. 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.

Consider an *Augmented Execution* scenario [17] in which a mobile device, hosting a NAM, is running short of a certain resource (e.g., battery power or CPU cycles), while a FM f is performing a demanding task (such as face recognition, video processing, or data mining). We expect an autonomic device with MCC support (e.g., a subscription to a cloud service) to react accordingly to the situation so that the task is completed successfully, although local resources are insufficient, and without requiring user intervention. In our scenario, a possible decision is to offload the demanding task for execution on the cloud service.

It is crucial to identify the responsibility of these decisions and the mechanisms to enact them. 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 maintain certain quality of service or safety conditions. Let us now start considering the role of policies of the functional module f in our specific example. Policies of f are $\langle P_o, P_l, P_r \rangle$, where:

$$P_o = \{(\mathsf{cpuLoadUpdate}, load > 70\%, \mathsf{offload}(\mathit{fid})),\\ (\mathsf{batteryChargeUpdate}, charge \leq 30\%, \mathsf{offload}(\mathit{fid}))\}$$

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

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

with fid being the identifier of f. On-site policies 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, P_l and P_r). The former monitors the quality of the wireless connection and decides (possibly, by enacting some forecasting) when it is necessary to request the module back because the connection has become unreliable and in order not to loose the computation performed so far. The latter resides on the guest 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 one or more virtual machines (VMs) *running a cloned system image*. The mobile device is allowed to offload f to a cloned replica for remote execution, thus saving battery and time, since the speedup factor of the cloud is higher. As already mentioned, the offloading process is started on the mobile device by policies of the functional module f. On the side of NAMs hosted on virtual machines, policies perform other monitoring tasks such as those described by the following rules:

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

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.

Figure 2 illustrates a possible interaction, among those allowed by the policies described previously. 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. Therefore, the virtual machine hosting nam_2 accepts module f (it may be running other modules). When offloading is complete, all the service requests on nam_1 that are dispatched to f are redirected to nam_2 for evaluation.

We assume that the VM 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

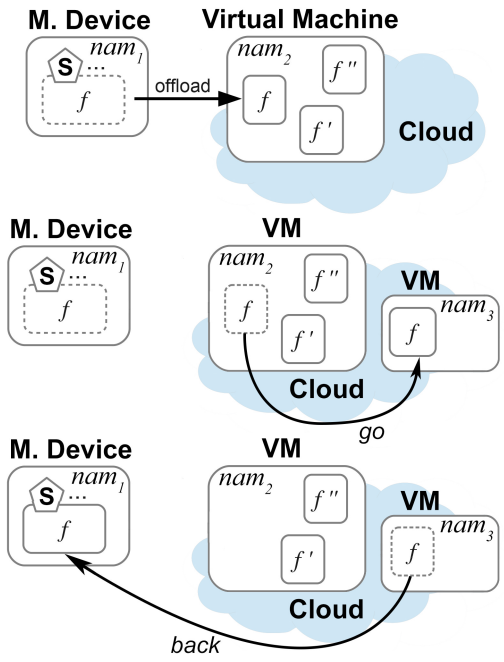

Figure 2. A possible evolution of the scenario described in the case study.

Figure 2, this balancing operation is illustrated as a go action moving f 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 remote policy handler, as discussed previously. Similarly, the virtual machine policy can also detect that the user has no residual credit and the functional module f cannot be hosted any longer. In both cases f must be sent back to the owner nam_1, so that the execution can continue locally. Also this event is illustrated in Figure 2, by showing that f goes from nam_3 back to nam_1, on the mobile device.

IV. 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 [13] 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.

Table I. 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*)	$a ::=$ **in**$(T)@\ell \mid$ **read**$(T)@\ell \mid$ **out**$(t)@\ell$
	$\mid$ **inp**$(T)@\ell \mid$ **readp**$(T)@\ell \mid$ **eval**$(P)@\ell$
	$\mid$ **newloc**$(s) \mid x := e$
(*Tuples*)	$t ::= e \mid \ell \mid P \mid t_1, t_2$
(*Templates*)	$T ::= e \mid \ell \mid ?x \mid ?l \mid ?X \mid T_1, T_2$

KLAIM syntax[1] is shown in Table I.

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).

[1] We use a version of KLAIM enriched with high-level features, such as assignments and standard control flow constructs (i.e., sequence, if-then-else, and while loop), that simplify the modeling task. Although these features were not included in the original presentation of KLAIM [10], they can be easily rendered with it (by resorting, e.g., to choice, fresh names and recursion in the usual way). The considered language is also equipped with the non-blocking versions of the retrieval actions, i.e. **inp** and **readp**. All the constructs mentioned above are directly supported by KLAIM related tools (such as, e.g., the analysis tool SAM [19]).

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, $\mathbf{inp}$ and $\mathbf{readp}$ act similarly to $\mathbf{in}$ and $\mathbf{read}$, and additionally return the value *true*; otherwise, they return the value *false* and the executing process does not block. $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space of the target node identified by ℓ, while action $\mathbf{eval}(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Both $\mathbf{out}$ and $\mathbf{eval}$ are non-blocking actions. Finally, action $\mathbf{newloc}$ 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.

V. 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 the this section we omit the target $\mathbf{self}$ from KLAIM actions, writing e.g. $\mathbf{in}(T)$ in place of $\mathbf{in}(T)@\mathbf{self}$.

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 $\rho_i = \{\mathbf{self} \mapsto nid_i\}$. Intuitively, each NAM $\langle nid, R, F, P\rangle$ is modelled 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 F, availability of resources in R, messages to denote service requests or events, code of functional modules in 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 P and executing the mobility commands;
- F_j includes the processes modeling the j-th functional module in 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.

A. NAM control

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

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

This process cyclically reads (and consumes) a service request, determines the NAM hosting the functional module implementing the service, 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:

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

Mobility commands are dealt with by the *mobility handler* (MH, illustrated in Sec. V-C), 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 P_n. In particular, an event ev (retrieved by an $\mathbf{in}$) triggers the execution of the processes P_{act}, corresponding to the action act, provided that condition co is satisfied.

B. FM control

Every FM 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:

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

On arrival of a service assignment (srvAssign) for *fid*, the service handler create a thread with 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:

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

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

After offloading, service requests are forwarded to the remote NAM nid_{DST}. 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 last behavior reacts to a go request (goSH) to nid_{DST} by creating a remote service handler to location nid_{DST} and terminating. Clearly, after a go action, service requests are forwarded to the new NAM, where the remote service handler is active.

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

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

By using the service identifier sid, the implementation ($Code$) of that service in the functional module *fid* is retrieved in a tuple tagged by 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. 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 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 identifier tid to unregister on completion and to react on migration/offloading. We assume 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 P_o of *fid*. Furthermore, it reacts to mobility actions identified by tuples with tag in {backPH,copyPH, goPH,migratePH,offloadPH}. In particular, similarly to the service handler, in the case of an offload request it first starts a remote policy handler RPH_{fid} (which executes the remote policy P_R) and then switches to execute a local policy handler LPH_{fid} (which executes the local policy P_L). Due to lack of space, we relegate the code of processes PH_{fid}, RPH_{fid} and LPH_{fid} to the Appendix.

C. 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.

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

On arrival of an offload request (offloadReq) the handler first informs the remote NAM nid_{DST} that its NAM (**self**) is the offload requester for 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. Finally, offload requests are sent to the service handler and to the policy handler by using offloadSH and offloadPH requests, respectively. 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, 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, 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, on arrival of a back request (backReq), performs two operations: (1) it sends back requests to the service and policy handler by using backSH and backPH, and (2) it sends to the remote NAM a go request with destination **self**.

Finally, the go action handler GH 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 (NAM2) so that it can update service bindings accordingly, (3) *moves* implementation and threads to the new location, (4) if the new destination NAM2 is the originator of the offload then it is indeed a back action, and it simply translates remote service assignments to local ones, otherwise it performs three sub-steps: (4.i) it informs NAM2 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**).

VI. Related Work

A. Mobile Cloud Computing

Many approaches to MCC have been proposed in the literature. In [17], 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 this work, we consider an Augmented Execution case study to illustrate our formalization. In the near feature we plan to extend our study to the other two approaches.

B. Autonomic Middleware

Autonomic Computing brings together many fields of computing, with the purpose of creating computing systems that manage themselves. MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) [15] is a reference model for an autonomic control loop.

Among available MAPE-K implementations, the Autonomic Computing Toolkit is a collection of self-managing autonomic technologies, allowing for the development of autonomic systems [20]. Also, the ABLE Toolkit [4] offers autonomic management in the form of a multi-agent architecture in which the *autonomic manager* is an agent or a set of agents. Kinesthetics eXtreme [16], [22] is an implementation of the MAPE-K loop, whose main purpose is the addition of autonomic properties to legacy systems.

NAM4J[2] is a Java middleware which has been specifically developed for implementing NAM-based autonomic systems. A layer stack showing the role of NAM4J in a networked system is depicted in Fig. 3. Basically, NAM4J runs on top of the operating system of a physical or virtual device (exploiting, of course, an appropriate Java Virtual Machine). In its turn, the middleware executes services provided by functional modules. Each functional module has an internal feedback loop to direct

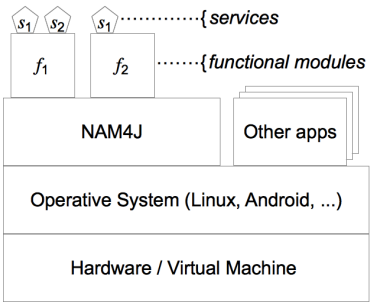

Figure 3. NAM4J Layer Stack

its objective-oriented behavior in an autonomic fashion, which is regulated by a set of *local* policies. Moreover, the NAM itself, i.e. the container of all modules, has a feedback loop for self-managing the overall unit, regulated by a set of *global* policies. We refer to Section II for a more complete account on NAMs, functional modules and services.

C. 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 [7]. Two possible scenarios exist: (1) *strong mobility*,

[2]NAM4J website: http://code.google.com/p/nam4j/

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. Some researchers addressed the problem at the application level, by using a pre-processor to filter the code, prior the execution, to insert statements having the purpose of capturing and re-establishing the state of Java threads [24]. Other solutions, such as JavaGo [23], provide source code translations. Cabri *et al.* proposed an approach for strong mobility on top of the IBM Jikes Research Virtual Machine (RVM) [6], by means of an extension of the IBM JikesRVM scheduler that Java programmers can dynamically enable, simply importing a Java package into their migrating Java applications. Unfortunately, this approach cannot be applied to applications running on mobile devices. On Android smartphones, for example, the Dalvik VM cannot be replaced by the JikesRVM (which specifically targets multiprocessor SMP servers). Other researchers chose to deal with Java strong mobility from the inside, by modifying the bytecode interpreter to keep track of the execution state [5]. Neither this approach can be applied to applications running on mobile devices. NAM4J currently supports only weak mobility — on both Java and Dalvik VMs. Anyway, in our formalization we have already considered both forms of mobility.

On the iOS platform, strong mobility is unfeasible, due to the SDK constraints imposed by Apple. However, for most practical applications which can benefit from MCC, weak mobility is sufficient. In this context, two models can be considered. With *Code Push*, mobile nodes discover suitable remote hosts and send them the code to be executed and the associated data. With *Code on Demand*, mobile nodes get the code, including data , from the Cloud or from other mobile nodes.

D. 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 [21], 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 Dπ, Djoin, KLAIM and Ambient, are surveyed and compared in [12].

Regarding autonomic computing, most of the proposals in the literature still concern full-fledged programming languages rather than foundational models. Some proposed formalisms, as e.g. in [2], [3], [25], are inspired by chemical and biological phenomena. A formalism closer to programming languages, following a process calculi approach and based on KLAIM, is SCEL [9]. Although it is equipped with constructs for dealing with autonomicity, SCEL 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.

In this paper we have selected KLAIM as basis for our formalization because, besides (strong and weak) mobility mechanisms, it also permits to model autonomic features conveniently (as shown in [14]). A combination of both mobility and autonomicity is necessary for proper modeling of MCC scenarios. On top of this, KLAIM comes with software tools that support various forms of analysis.

VII. 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 formal approach at work on a realistic case study, including not only offloading but also other cost- and reliability-driven strategies.

This work is a first step for some research lines we envisage. 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 [19]. The challenge here is the identification of relevant and desirable properties for MCC. In particular, the stochastic extension of KLAIM [11], 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. This would support effective decision making in mobility strategies. NAM4J could be used to extract information from execution traces, to determine the appropriate parameters for the stochastic models.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] M. Amoretti, M. Picone, and F. Zanichelli. Global Ambient Intelligence: An autonomic approach. In *Proc. of PerCom Workshops*, pages 842–847, 2012.

[2] O. Andrei and H. Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought*, volume 5420 of *LNCS*, pages 15–26. Springer, 2009.

[3] J.-P. Banâtre, Y. Radenac, and P. Fradet. Chemical Specification of Autonomic Systems. In *IASSE*, pages 72–79. ISCA, 2004.

[4] J. P. Bigus, D. A. Schlosnagle, J. R.Pilgrim, W. N. Mills III, and Y. Diao. ABLE: a toolkit for building multiagent autonomic systems. *IBM Systems Journal 41*, (3):350–371, 2002.

[5] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer. Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. In *I.N.R.I.A., Research report n.4662*, December 2002.

[6] G. Cabri, L. Leonardi, and R. Quitadamo. Enabling Java Mobile Computing on the IBM Jikes Research Virtual Machine. In *Proc. of PPPJ*, 2006.

[7] A. Carzaniga, G.P. Picco, and G. Vigna. Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. In *Proc. of ICSE*, 2007.

[8] D. Da Silva. 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.

[9] R. De Nicola, G. L. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *FMCO*, volume 7542 of *LNCS*, pages 25–48. Springer, 2012.

[10] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[11] R. De Nicola, J.P. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.

[12] G. L. Ferrari, R. Pugliese, and E. Tuosto. Calculi for Network Aware Programming. In *WOA*, pages 23–28. Pitagora Editrice Bologna, 2000.

[13] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[14] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi. Modeling adaptation with a tuple-based coordination language. In *SAC*, pages 1522–1527. ACM, 2012.

[15] IBM. An architectural blueprint for autonomic computing. Technical report, June 2005. Third edition.

[16] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics eXtreme: An external infrastructure for monitoring distributed legacy systems. In *Proc. of AMS*, pages 22–30, 2003.

[17] D. Kovachev and R. Klamma. Beyond the client-server architectures: A survey of mobile cloud techniques. In *Proc. of ICCC*. IEEE, 2012.

[18] K. Kumar and Y-H. Lu. Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? *IEEE Computer, Vol. 43, Issue 4*, April 2010.

[19] M. Loreti. SAM: Stochastic Analyser for Mobility, 2010. Available at http://rap.dsi.unifi.it/SAM/.

[20] B. Melcher and B. Mitchell. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technology Journal 8*, (4):279–290, 2004.

[21] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[22] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting autonomic capabilities onto legacy systems. *Tech. Rep. CUCS-026-03, Columbia University*, 2003.

[23] T. Sekiguchi, A. Yonezawa, and H. Masuhara. A simple extension of java language for controllable transparent migration and its portable implementation. In *Proc. of Coordination*, 1999.

[24] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for Transparent Thread Migration in Java. In *Proc. of MA*, September 2000.

[25] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In *SAC*, pages 295–302. ACM, 2012.

VIII. APPENDIX

For reviewers' convenience, in this appendix we report and briefly comment the entire KLAIM specification of NAM formalization. We start by reviewing the various kinds of tuples used to synchronize the NAMs and to realize mobility actions.

A. 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 $Proc$:

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

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:

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

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:

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

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

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

When the mobility handler reacts to mobility requests it sends appropriate mobility commands to the service handler of the corresponding FM:

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

and to its policy handler:

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

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

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

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

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

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

B. NAM control

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

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

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

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

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

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:

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

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, the threads, and sends migrate requests to the service and policy handler:

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

The offload action handler has been discussed in the paper:

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

The back action handler BH, on arrival of a back request, sends back requests to the service and policy handler, and sends to the remote NAM a go request with destination $\mathbf{self}$:

$BH =$
  $\mathbf{in}(\mathsf{backReq}, ?fid, ?nid_{DST});$
    $\mathbf{out}(\mathsf{backSH}, fid, nid_{DST});$
    $\mathbf{out}(\mathsf{backPH}, fid, nid_{DST});$
    $\mathbf{out}(\mathsf{goReq}, fid, \mathsf{self})@nid_{DST};$
    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, *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 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, $\mathbf{self}$).

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

C. FM control

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

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

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

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

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

Similarly 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:

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

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

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

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

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

D. 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 $\mathbf{in}/\mathbf{read}$ 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:

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

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

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

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:

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

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

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

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

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:

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

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

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

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

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

Moving threads of a functional module during offloading or migration is performed by retrieving (and deleting) each thread identifier associated to 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.