Towards a Flexible Middleware for Context-aware Pervasive and Wearable Systems

(Invited Paper)

Marco Muro and Michele Amoretti and Francesco Zanichelli and Gianni Conte
Dept. of Information Engineering
University of Parma
Via Usberti 181/a, 43124 Parma, Italy
Email: mmuro@ce.unipr.it

***Abstract*—Both ambient intelligence and wearable computing do need novel hardware and software, yet another enabling factor is *middleware*, which should be highly capable, flexible and efficient, allowing for the reuse of existing pervasive applications when developing new ones. In the considered application domain, middleware should also support quick runtime re-configuration of systems, interoperability among different platforms, efficient communications, and context awareness. In the on-going "everything is networked" scenario scalability appears as a very important issue, for which the peer-to-peer (P2P) paradigm emerges as an appealing solution for connecting software components in an overlay network, allowing for efficient and balanced data distribution mechanisms. In this paper we go through the state of the art of middleware for pervasive and wearable computing, and we outline our approach that is geared to autonomic systems and based on the P2P paradigm as well as on dynamic refactoring of software entities.**

I. Introduction

The concept of ambient intelligence (AmI) refers to environments that are aware and responsive to the presence of people, proactively supporting them in their daily lives. AmI overlaps with other ICT research topics, such as pervasive computing, networked embedded systems and artificial intelligence. Very near to AmI, wearable computing hopes to produce a paradigm shift of how a computer should be used. A person's computer should be worn, much as eyeglasses or clothing are worn, and interact with the user based on the situational context.

Both AmI and wearable computing need novel hardware and software, yet they also call for a highly flexible and efficient *middleware* allowing for reusing of existing pervasive applications, integrated with new ones. Middleware for pervasive systems should support quick runtime re-configuration of systems, interoperability among different platforms, efficient communications, and context awareness. In the on-going "everything is networked" scenario scalability appears as a very important issue, for which the peer-to-peer (P2P) paradigm emerges as an appealing solution for connecting software components in an overlay network, allowing for efficient and balanced data distribution mechanisms - that are crucial in pervasive applications, being them mostly based on the publish-subscribe paradigm, with hundreds of potential context event providers and consumers.

In the last decade the P2P paradigm has emerged as a highly appealing solution for scalable and high-throughput resource sharing among decentralized computational entities, by using appropriate information and communication systems without the necessity for central coordination [9]. In distributed systems based on the peer-to-peer paradigm all participating processes have the same importance. In contrast with the client/server approach, in which resource providers and resource consumers are clearly distinct, on the contrary peers usually play both roles. Furthermore, a peer-to-peer system is a complex system, because it is composed of several interconnected parts that as a whole exhibit one or more properties (*i.e.* behavior) which are not easily inferred from the properties of the individual parts [8].

The P2P approach is one of the main tenets of our research. Currently we are focusing on *autonomic* P2P systems, being able to detect, diagnose and repair failures and adapt their behavior to changes in the environment - which is highly appealing for a wide range of pervasive systems. We argue that, by increasing the context-awareness of monitoring data exchanged by autonomic peers, it is possible to efficiently sense network conditions and the level of provided services and perform corrective actions. Sharing context-based information can be realized through dissemination of specific data among different nodes or through cross-module and cross-layer messages inside the same node. For example, a QoS entity responsible to allocate network resources may exchange context-aware information with other nodes in order to identify changes in the network conditions.

Thus, in this paper we outline the state of the art of middleware for pervasive and wearable computing (in section II), and we introduce our approach that is geared to autonomic systems and based on the P2P paradigm as well as on dynamic refactoring of software entities. In section III we illustrate our reference model, called **Networked Autonomic Machine (NAM)**, for the characterization of networks whose nodes are provided with functional modules and services that are allowed to migrate. In section IV we illustrate the implementation choices and strengths of the NAM-based middleware we are developing, called **nam4j**. Finally, in section V we conclude the paper outlining future work.

II. State of the Art

The development of pervasive and wearable computing software is a complex and error-prone task because it must cope with heterogeneous infrastructures and with system dynamics in an open network. Middleware role is therefore essential to support mobility and adaptation of applications to the current context [4], [6], [10]. Literature provides a wealth of approaches, for which it is not possible to cover them all here. We limit the discussion to a couple of important and recent european projects, namely HYDRA and PERSONA, as well as two research works that take into account autonomicity and code refactoring, besides the common idea of an extensible middleware supporting service discovery, distributed information sharing, context management.

The HYDRA project [3] develops middleware for networked embedded systems that allows developers to create ambient intelligence applications utilizing device and sensor networks. The proposed middleware provides support for embedding services in devices and for proxying services for devices. Moreover, middleware supports dynamic reconfiguration and self-configuration. The dynamic context is modeled with runtime concepts and properties in four OWL ontologies defined within the HYDRA project: StateMachine, Malfunction, QoS, and Device. The Semantic Web Rule Language (SWRL) is used to define complex context events, such as "the user is far away from home", that can be used to take actions, like "surveillance system is switched to the highest security level".

Another remarkable solution is that implemented within EU project PERSONA [1], consisting in an architecture based on a set of different communication buses, each of them adopting specific and open communication strategies. Its design has been driven by the need for a a self-organizing infrastructure allowing the extensibility of component/device ensembles in an ad hoc fashion. In order to achieve this goal, the PERSONA Middleware implements distributed coordination strategies that provide the necessary service discovery, service orchestration and service adaptation functionalities. Components linked with the PERSONA Middleware may register with some of these communication buses, find each others and collaborate trough the local instances of the buses. Input and output buses support multi-modal user interactions with the system. The context bus is an event-based channel to which context sources are attached. Published events may be re-elaborated and transformed in high level events (situations) by components that have subscribed to the bus (*e.g.* context reasoners). The service bus is used to group all the services available in the AAL-space, being them atomic or composite (whose availability is managed by a Service Orchestrator component).

With respect to code refactoring, we find the *SelfLet* approach proposed by Devescovi *et al.* [2] very appealing. It does target autonomic systems whose nodes (*i.e.* the SelfLets) may change their structure at run time in order to acquire new capabilities (by dynamically loading software components that implement specific *abilities*), or to lose their capabilities (by undeploying components). Each SelfLet has a basic behavior that implements the SelfLet life cycle, specified as a set actions to perform, by a set of Goals to use or achieve, and by a set of autonomic rules supporting application dependent self-configuration. Each SelfLet has a knowledge base, for which adaptation is epigenetic, *i.e.* based on learning and knowledge transmission. Conversely, we are currently working on the problem of adaptive re-structuring of peers using a phylogenetic approach, *i.e.* memoryless transformations.

Another interesting approach for peer restructuring has been proposed by Tyson *et al.* [12], that show how *survival of the fittest* has been implemented into the Juno middleware. On receipt of a superior component, Juno dynamically reconfigures the internal architecture of the peer, by replacing the existing component with the new one. For example, a node would evolve its maintenance algorithms by obtaining a new maintenance component that offers the correct interface. The old component would then be removed from the software architecture and replaced by the new one. All subsequent maintenance functionality would then be performed by the new component. To support this, interchangeable components must offer identical interfaces. Moreover, to evaluate a new component, all meta-values are defined relative to the *default* component that defines a base-line for all equivalent components. This approach removes the necessity for other components and applications to possess semantic knowledge of quantitative values. Instead, it is possible to simply consider their capabilities as relative to each other.

III. Networked Autonomic Machines

In this section we introduce our formal tool called Networked Autonomic Machine (NAM), that allows to model a hardware/software system able to

- communicate with other NAMs;
- execute a number of functional modules that may provide/consume services or context events;
- dynamically deploy, undeploy and migrate functional modules and services.

These features support local self-management and self-healing activities. Conversely, the achievement of global autonomicity, in a system of NAMs, depends on the policies adopted at the functional level. For example, by providing all NAMs with a peer-to-peer overlay management module, it may be possible to enable their cooperation in routing messages for discovering new services or functional modules to use or download and execute.

NAM is suitable for modeling several kinds of computing machines: PCs and workstations, notebooks, PDAs, smartphones, as well as sensors and actuators. NAMs can be of different types and complexities, depending on their resources (OS, processor type, memory, I/O type, battery, connectivity) and functionalities (*e.g.* communication protocols, processing features, sensing capabilities).

With respect to other modeling tools, NAM allows to specify the migration of functional modules and services among nodes. The NAM formalism can be used to semantically characterize and compare the elements of a self-*, highly dynamic distributed system.

A. Formal Description

Formally, a NAM node is a tuple

$$NAM = \langle R, F \rangle$$

where R is a set of physical *resources*, such as CPU cycles, storage, bandwidth, and F is a set of *functional modules*.

Each functional module $f \in F$ plays one or more of the following roles:

- context provider (CP),
- context consumer (CC),
- service provider (SP),
- service consumer (SC).

The formal definition is

$$f = \langle S_f, S_r, C_{in}, C_{out}, POL \rangle$$

where S_f is the set of provided services, S_r is the set of services the module can consume, C_{in} is the set of consumed context events, C_{out} is the set of provided context events, and POL is the set of policies according to which the functional module can react to the sensed environment. More precisely, when receiving a set of context events in C_{in}, the functional module may react by publishing a set of context events in C_{out}, by executing a set of services in S_f, or by calling a set of services in S_r. In section III-B we discuss about automatic evaluation of policies within functional modules.

A service consists in a unit of work executed by a service provider to achieve the results desired by a service consumer. In general, a *service* is a tuple

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

where I is a set of input parameters, and O is a set of output parameters. Each I/O parameter has a type, *i.e.* a class (still using the ontological terminology). It is important that service consumers and service providers share the same domain ontologies in order to have a common understanding of shared services. Semantic descriptions of services are used to organize service advertisements in centralized or distributed repositories, allowing to efficiently retrieve and use services in the network. P and E are the precondition and effect sets, respectively. Such optional parameters are expressed in the form of logical conditions which can assume the true or false value. Preconditions must be verified in order to invoke the service, while an execution effect may become a precondition for the successive invocation in a composition scenario. For example, in an ambient intelligence scenario, a service for switching lights may be simply defined by the effect of assigning the value ”ON” to the ”status” property of a ”living room light”. The IOPE approach is adopted for example in the OWL-based Web service ontology called OWL-S [5].

An *atomic service* is defined as the minimal executable function unit, that cannot be decomposed and whose execution can transform a given state to another state. It is represented as a tuple:

$$as = \langle I, O, P, E, Q \rangle$$

where Q is the set of exposed quality-of-service (QoS) properties, such as reliability, availability, performance, security and timing. Each node can provide different atomic services. The number of concurrent service instances and the quality of service (QoS) of each instance at a certain time depends on the current availability of hardware resources on the node.

Intra- or inter- NAM services can be statically or dynamically aggregated (proactively or on-demand) to realize new complex tasks. A *composite service* is a tuple:

$$cs = \langle I, O, P, E, Q, G_w \rangle$$

where G_w is the rule that allows to combine atomic services; this rule is represented as a directed workflow graph

$$G_w = \langle S_w, L_w \rangle$$

where S_w is a set of services (both atomic and composite) and L_w is a set of links that represent transitions (*i.e.* *I-O* connections) among services.

Figure 1 shows the internal structure of a NAM, with resources, functional modules, and services.

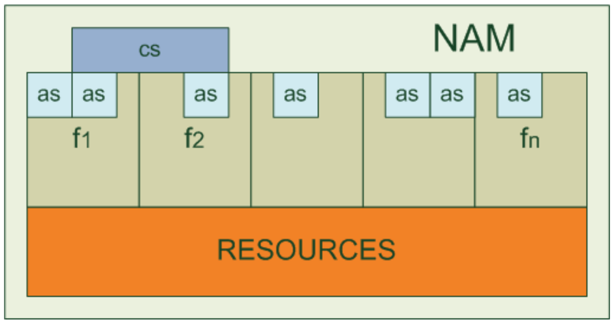

Fig. 1. The internal structure of a NAM. Functional modules use resources and provide atomic services (as), that can be aggregated in order to provide composite services (cs).

NAMs are able to dynamically reconfigure their structure, by adding new functional modules or services, or discarding those that are no more necessary. The *migration* of functional modules and services is another NAM's feature defined as

$$m_f = \langle NAM_u, NAM_d, f \rangle$$

$$m_s = \langle NAM_u, NAM_d, s \rangle$$

where $f \in F$ is a functional module, $s \in S$ is a service, NAM_u is the uploader and NAM_d is the downloader. A migration can start only if the resources of the downloader match with the set R_m of resources (*e.g.* running environment, libraries, CPU, RAM, etc.) required to run the functional module or the service to be migrated.

B. Autonomic Reasoning in NAM

Policies are the central means of control for autonomic computing systems. They consist of goals, scope, constraints according to which the system should produce behavior that users find reasonable. In the NAM perspective, a functional module may be provided with a more or less sophisticated decision engine that, using a set of rules, is able to derive new facts (output) from known facts (input). Policies can be statically defined or dynamically learnt.

Facts and rules should be defined according to an *ontology* shared among NAMs. The ontological approach tries to reduce the gap between informal language and programming language by means of formal ontology languages. Formal ontology languages have a precise semantics that allow for unambiguous specifications of domain knowledge. These languages are easy enough to learn quickly (much faster than programming languages), not just by software engineers, but also by domain experts.

Figure 2 illustrates a functional module that interact with its environment, that may include local modules and remote NAMs, but also anything outside than the NAM network (*e.g.* operators, end users, the weather, etc.). Interaction with the environment is based on two pillars: perception, that means context-awareness, and decision, which is the (policy-based) ability to infer new context events, service calls, or output (to some users, or to a service request).

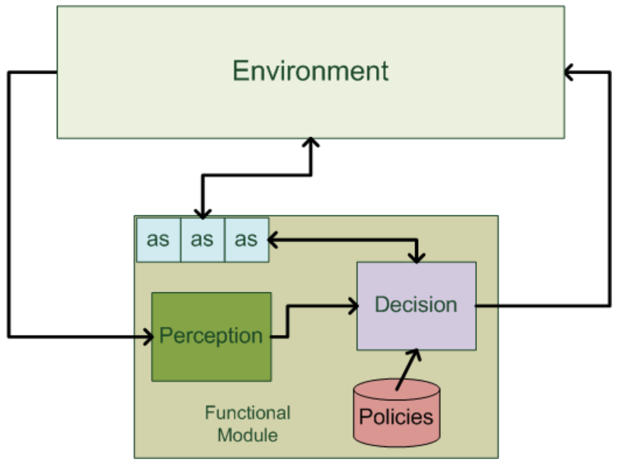

Fig. 2. A functional module that consumes context events (perceives the environment) and, based on policies, decides how to interact with the environment.

There are a variety of languages which can be used for representation of conceptual models, with varying characteristics in terms of their expressiveness, ease of use and computational complexity [11]. Normann and Putz suggest to use Prolog to define ontologies for ambient intelligence systems [7]. We argue that Prolog may be applied to any autonomic distributed system, once provided with mechanisms for checking the consistency of fact and rules. Indeed, two main issues must be considered. First, a recently consumed context event (fact) may supersede a previous one, or not. For example "Jack is sitting", received after "Jack is walking", should replace the latter in the knowledge base of the NAM. On the other side, "Jack is talking" may be considered as supporting information, not replacing "Jack is walking". Second, the set of rules may change over time, taking care that new rules do conflict with existing ones.

In our framework, two connected NAMs can interact if and only if they share the same ontology, or at least if their ontologies partially overlap. The two NAMs may use any reasoning module they like (being it Prolog-based, or not).

IV. The design of NAM4J

In this section we illustrate nam4j, which is a Java implementation of the NAM framework. nam4j provides interfaces and abstract classes for developing functional modules and NAMs (as illustrated in figure 3).

Figure 3 illustrates the structure of a generic functional module, in terms of Java classes.

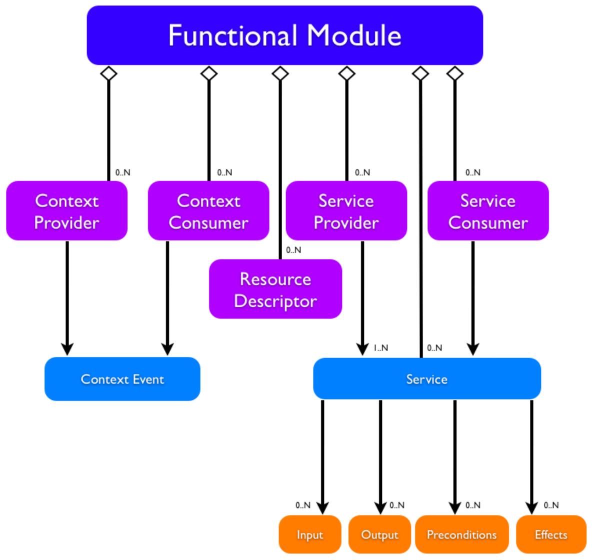

Fig. 3. Structure of a generic functional module.

A generic functional module may be:

- context provider (CP): providing tools to catch context events to a generic environment (es. sensors or PMD) and to forward in the system;
- context consumer (CC): catching and managing context events existing in the network;
- service provider (SP): providing instruments to manage a specific service following NAM model in which a service is a tuple $\langle I, O, P, E \rangle$;
- service consumer (SC): providing the use of an existing service in the network;

As an example in a e-health scenario, we consider a subnetwork with four NAMs, each one being characterized by one of the following functional modules:

- f_{brac} = [CP], used by NAM-1 that runs in a bracelet used to generate context events depending of the evaluation of level blood pressure of the user by sensors (a low level or high level generate an event context);

- f_{cam} = [CP], used by NAM-2 that is embedded in a camera that generates context events describing the posture of the user (standing up, lying down or sit down);
- f_{AM} = [CC,SC], used by NAM-3 as monitor that catches context events generated by NAM-1 and NAM-2 and - if necessary - invokes a service on NAM-4 to start an emergency call; it includes a Prolog-based logic reasoner;
- f_{call} = [SP], used by NAM-4 to provide a service for emergency calls.

Every NAM is equipped with an overlay module (f_{over} = [SC,SP]) to communicate and to discover nodes in the local network. Such a module can implement a specific P2P protocol, in order to provide a common overlay network for the communication of each other module using the features of the selected P2P scheme.

Sensor-based NAMs in the bracelet and in the camera periodically analyze the environment. If the NAM detects a change of the state in the measured variables, it generates a specific context event. NAM-3 listens on context bus waiting for context events: if a user has low blood pressure and she/he is lying down, NAM-1 and NAM-2 generate context events that are caught by NAM-3, whose functional module f_{AM} detects the danger and starts an emergency call with no loss of time. We assume the availability of an intelligent camera able to recognize three different postures for the user (laying down, sitting and standing) and the bracelet monitoring in a periodic way two important health parameters, systolic blood pressure and diastolic blood pressure.

We chose to use Prolog to define formal concepts of the ontology as well as rules, in particular by building the decision engine on tuProlog, a Java-based lightweight Prolog reasoner for Internet applications and infrastructures. Moreover, it provides a straightforward API to implement simple or more complex Prolog program within Java code, or to read existing Prolog expressions from a file or from a database. Once one or more Prolog theories have been acquired, it is possible to use them to evaluate facts and derive new facts.

According the previous example we have defined a theory based on five predicates, using Prolog:

```
emergencyCall(X) :- laying(X),
                    (lowPressure(X);
                     highPressure(X)).
lowPressure(X) :- measuredPressure(X,Y),
                  limitLowPr(Y).
highPressure(X) :- measuredPressure(X,Y),
                   limitHighPr(Y).
limitLowPr(A) :- A < 70.
limitHighPr(B) :- B > 120.
```

That is, emergencyCall for a generic user X is true only if X is laying and lowPressure or highPressure is true, where lowPressure for X is true only if the measured blood pressure exceeds the limitLowPr definition (the same applies for high blood pressure). NAM-1 and NAM-2 generate context events like:

```
laying(user).
measuredPressure(user,121).
```

When NAM-3 catches such context events, it performs the following Prolog query:

```
?- emergencyCall(user).
```

If the engine gives a positive answer, NAM-3 provides to start an emergency call invoking the service exposed by NAM-4. Of course, the language allows for the definition of more complex and realistic conditions for triggering an emergency call, as well as policies for dealing with node failures.

V. Conclusion

In the paper we analyzed the state of the art of middleware for pervasive and wearable computing, and we proposed the formal concept of Networked Autonomic Machine (NAM) which we are concretizing within the nam4j middleware. As future work, we are going to complete the development of nam4j along with a number of significant demos related to the AmI application domain. We are focusing on the challenging problem of creating dynamic knowledge bases for NAMs, taking into account the temporal dimension while defining policies.

References

[1] A. Fides-Valero, M. Freddi, F. Furfari, M.-R. Tazari, *The PERSONA Framework for Supporting Context-Awareness in Open Distributed Systems*, Proc. of AmI 2008, Nurnberg, Germany, November 2008.

[2] D. Devescovi, E. Di Nitto, D. Dubois, R. Mirandola, *Self-Organization Algorithms for Autonomic Systems in the SelfLet Approach*, Proc. of the 1st Int'l Conference on Autonomic computing and communication systems, Rome, Italy, October 2007.

[3] M. Eisenhauer, P. Rosengren, P. Antolin, *A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence System*, Proc. of the IEEE Int'l Workshop on Wireless Ad-hoc and Sensor Networks (IWWAN), Rome, Italy, June 2009.

[4] Z. Jaroucheh, X. Liu, S. Smith, *A Perspective on Middleware-Oriented Context-Aware Pervasive Systems* Proc. of the 33rd Annual IEEE Int'l Computer Software and Applications Conference, Seattle, Washington, USA, July 2009.

[5] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara, *Bringing Semantics to Web Services: The OWL-S Approach*, Proc. of the 1st Int'l Workshop on Semantic Web Services and Web Process Composition (SWSWPC), San Diego, California, USA, July 2004.

[6] C. Narayanaswami, *Middleware for Wearable Computing*, In Handbook of Mobile Middleware, by P. Bellavista and A. Corradi, Auerbach Publications, 2006.

[7] I. Normann, W. Putz, *Ontologies and Reasoning for Ambient Assisted Living*, AALiance Conference, Malaga, March 2010.

[8] J. M. Ottino, *Engineering complex systems*, Nature, 427, 399, 2004.

[9] D. Schoder, K. Fischbach, *Peer-to-Peer Paradigm*, Proc. 37th IEEE Int'l Conference on System Sciences, Hawaii, USA, 2004.

[10] Y. Song; S. Moon; S. Gyudong; D. Park, *mu-ware: A Middleware Framework for Wearable Computer and Ubiquitous Computing Environment*, PerCom Workshops '07, Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, March 2007.

[11] R. Stevens et al., *Ontology-based knowledge representation for bioinformatics*, Briefings in Bioinformatics 2000 1(4):398-414; doi:10.1093/bib/1.4.398

[12] G. Tyson, P. Grace, A. Mauthe, S. Kaune, *The Survival of the Fittest: An Evolutionary Approach to Deploying Adaptive Functionality in Peer-to-Peer Systems*, Proc. of the 7th workshop on Reflective and adaptive middleware, Leuven, Belgium, December 2008.