# A survey about context-aware middleware

Marco Bessi
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
marco.bessi@mail.polimi.it

Leonardo Bruni
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
leonardo1.bruni@mail.polimi.it

## ABSTRACT

Context-aware systems represent extremely complex and heterogeneous systems. The need for middleware to bind components together is well recognized and many attempts to build middleware for context-aware system have been made. One of the goal of this paper is to provide a general introduction about the evolution of the middlewares and than to proceed with an analysis of the requirements and the issues for context-aware middleware. The paper also provides a survey of some approach for this new kind of middleware.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*;
C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

## Keywords

Middleware, context, context-awareness, reflection, ontology, tuple-space, publish-subscribe

## 1. INTRODUCTION

Under the highly variable computing environment conditions that characterize mobile platforms, it is believed that existing traditional middleware systems are not capable of providing adequate support for the mobile wireless computing environment. There is a great demand for designing modern middleware systems that can support new requirements imposed by mobility; therefore, it is increasing the importance of context-awareness in distributed applications. Context-aware applications adapt its behavior to changes in the environment and user requirements. The complexity of developing context-aware applications makes middleware an essential requirement.

In this paper, we evaluate the current state-of-the-art in middleware for context-aware applications. The structure of the paper is as follows. In section 2, we characterize middleware and the evolution to its context-aware solution. In 2.4 and 2.5, we analyze the requirements that context-aware middleware should meet and the issues that we have to handle. Than, in section 3, we review some approaches to context-aware middleware and, for each one, we present an existing model.

## 2. THE ROAD TO A "NEW GENERATION" MIDDLEWARE

There are many different definition of "middleware" in the literature; however we can assert that it plays an important role in hiding the complexity of distributed applications. These applications typically operate in an environment that may include heterogeneous computer architectures, operating systems, network protocols, and databases. Middleware's primary role is to conceal this complexity from developers by deploying an isolated layer of APIs.

Middleware is defined as follows by *Linthicum* [16].

> *"Middleware is an enabling layer of software that resides between the application program and the networked layer of heterogeneous platforms and protocols. It decouples applications from any dependencies on the plumbing layer that consists of heterogeneous operating systems, hardware platforms and communication protocols"*

### 2.1 Traditional middleware

Existing middleware technologies have been built adhering to the metaphor of the *black box*. They succeeded in hiding away many requirements introduced by distribution, such as the heterogeneity and fault-tollerance, offering them an image of the distributed system as a single integrated computing facility. In other words, the presence of a middleware to build distributed systems free developers from the implementation of low-level details related to the network, like concurrency control, transaction management, network communication, in such a way they can focus on application requirements.

It can be observed that the use of traditional middleware, conceived for fixed distributed systems and on the idea of a static context, in such complex and heterogeneous environments is not always feasible (see figure 1). Traditional distributed systems assume a stationary execution environment that contrast with the extremely dynamic scenario of

| | Distributed environments | Mobile environments |
|---|---|---|
| bandwidth | high | low |
| context | static | dynamic |
| connection type | stable | unstable |
| mobility | no | yes |
| communication | synchronous | asynchronous |
| resource availability | high | low |

**Figure 1: Main differences between distributed and mobile environments.**

the context-aware computing. Middlewares used in such systems hide low-level network details reaching a high level of transparency to applications. Instead, in mobile environments the context is extremely dynamic and it cannot be managed by a priori assumptions. Then, it is mandatory to implement re-configuration techniques able to react to the changes in the operating context.

## 2.2 Context-aware middleware

Modern distributed applications need a middleware that is capable of adapting to environment changes.

Ngo *et al.* [15] defines context as:

> *"Context is any information about the circumstances, objects, or conditions by which a user is surrounded that is considered relevant to the interaction between the user and the ubiquitous computing environment".*

Context-awareness involves acquisition of contextual information, reasoning about context and modifying one's behavior based on the current context. A middleware for context-awareness would provide support for each of these tasks. It would also define a common model of context, which all agents can use in dealing with context. It would also ensure that different agents in the environment have a common semantic understanding of contextual information.

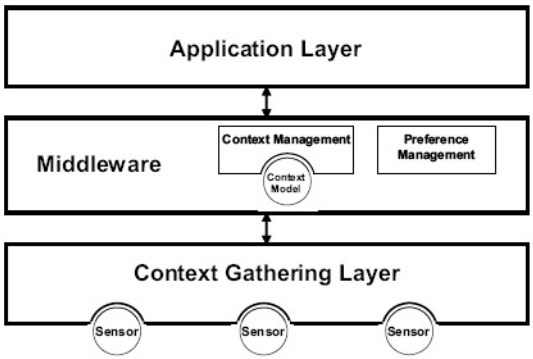


**Figure 2: Architecture of a context-aware system.**

## 2.3 Is it necessary?

Now, a question came out: is it very necessary built an intermediate layer for the context management or we can manage the context directly on the applications or at operating system level?

Different approaches have been suggested for promoting context-awareness among agents. An approach is developing an infrastructure or a middleware for context-awareness [21]. A middleware would provide uniform abstractions and reliable services for common operations. It would simplify the development of context-aware applications. A middleware would simplify the tasks of creating and maintaining context-aware systems. It would also make it easy to incrementally deploy new sensors and context-aware agents in the environment. A middleware would be independent of hardware, operating system and programming language. Finally, a middleware would also allow us to compose complex systems based on the interactions between a number of distributed context-aware agents.
Adaptation at operating system level is platform-dependent and requires a deep knowledge of the internals of the operating system. In addition, unwanted changes at this level could be catastrophic.
Adaptation at the application level imposes an extra-burden to the application developer and the adaptation mechanisms developed at this level cannot be reused since they are application specific.
In addition, both adaptation at operating system and adaptation at application level nullify the advantages listed above. This leads us that adaptations should be handled at middleware level.

## 2.4 Requirements

The middleware we are considering must address many of the requirements of traditional distributed systems, such as heterogeneity, mobility, scalability and tolerance for component failures. In addition, it must protect user's privacy. The large number of components that are present in context-aware systems introduces a requirement for straightforward techniques for deploying, configuring and managing networks of sensors [12] [14].

- **Support for heterogeneity:** hardware components ranging from resource-poor sensors, actuators and mobile client devices to high-performance servers must be supported, as must a variety of networking interfaces and programming language.
- **Support for mobility:** all components can be mobile and the communication protocols must therefore support appropriately flexible forms of routing. Context information may need to migrate with context-aware components.
- **Scalability:** context processing components and communication protocol must perform adequately in very changing domains.
- **Support for privacy:** flows of context information between the distributed components of a context-aware system must be controlling according to user's privacy needs and expectations.
- **Tolerance for component failures:** sensors are likely to fail in the ordinary operation of a context-aware system; disconnection may also occur.

- **Ease of deployment and configuration:** it must be easily deployed and configured to meet user and environmental requirements.
- **Dynamic reconfiguration:** detecting changes in available resources and reallocating them or notify the application to change its behavior.
- **Adaptivity:** the ability of a system to recognize unmet needs within its execution context and to adapt itself to meet those needs.
- **Asynchronous paradigm:** decoupling the client and server components and delivering multicast messages.

## 2.5 Issues

With the birth of context-aware middleware many issues are came out, like security, balance of user control, sensing the context and conflicts. In the following sections we examine the balance of user control and the conflict problem about the new middleware approach.

### 2.5.1 Balance of user control

In order to increase software autonomy, applications depend on context information to dynamically adapt their behavior to match the environment and user requirements. Therefore, context-aware applications not only require middleware for distribution transparency of components, but also to support personalization and adaptation based on context-awareness. However, context-aware applications may not always adapt as the user expects, and may cause users to feel loss of control over the behavior of their applications [13].

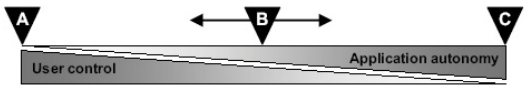


**Figure 3: Continuum of user control versus software autonomy**

How we can see in figure 3, at the leftmost end (A), users are given full control over application behavior, and applications have very little autonomy. Applications designed in this way are the most interactive. Conversely, at the other end (C), applications only require a small amount of user control. Applications can also occupy any intermediate position on the continuum (position B). The appropriate position along the continuum will be dictated by the userŠs needs, situation and expertise.

In traditional applications, the trade-off between user control and software autonomy has been fixed at design-time. In contrast, context-aware applications may need to adjust the balance of user control and software autonomy at run-time by adjusting the level of feedback to users and the amount of user input.

### 2.5.2 Conflicts

Applications dynamically change the set of resource that middleware monitors on their behalf, the context configurations they are interested into, and the behaviors they want to adhere to. While doing so, applications may introduce ambiguities, contradictions and logical inconsistencies. We refer to these inconsistencies as *conflicts* [2].

When setting up application profiles, the following two basic typologies of conflicts may be created.

- **Intra-profile conflict:** a conflict exists inside the profile of an application running on a particular device. It identifies conflicts that are local to a middleware instance.
- **Inter-profile conflict:** a conflict exists between the profiles of applications running on different devices. It identifies conflicts that are distributed among various middleware instances.

As examples for intra- and inter-profile conflicts we can use the conference scenario.
About *intra-profile conflict*, we assume that the *talk reminder* service can be delivered using the following policies: *silentAlert* policy (battery<15), (location=indoor) or *vibraAlert* policy (location=indoor). Each of these policies require different amounts of resource to be used. Whenever a talk reminder service has to be delivered, the application profile is consulted to find out which policy to apply. Lets us assume battery lower than 15 and the service is invoked indoor, so are enabled both the policy.
About *inter-profile conflict*, we assume that peer can *exchange messages* using one of the following policies: *plainMsg* policy or *encryptedMsg* policy. Lets suppose that two peer want to start a chat. So, they have to agree on a common policy to be applied to exchange messages. During the lifetime of that chat the policy used may change to adapt to the new context. However, all the chatting peers must agree on the new policy to use. But, the context of the two peer are different so, they can have a conflict between the policies.

**Conflict resolution mechanism**
Whenever a service that incorporates a conflict is requested, a conflict resolution mechanism has to be run to solve the conflict and find out which policy to use to deliver the service, otherwise application cannot execute. To design this mechanism, the following requirements have to be considered.

- **Dinamicity:** never intra- nor inter-profile conflicts can be detected and resolved statically. Due to the complex nature of context, a stati conflict analysis would produce an explosion in the context information that must be checked, and would require a consumption of resources that portable devices can not bear. An external service on a powerful machine that is contacted on-demand is not feasible either because it would require a persistent connectivity (not granted). As consequence, a dynamic solution is needed.
- **Simplicity:** the conflict resolution mechanism must be simple. It must not consume resources that are already scarce on a mobile device.

- **Customization:** on one hand, middleware should be in charge of carrying out the conflict resolution process in an automatic way as much as possible. On the other hand, it must be possible for the applications to customize the conflict resolution mechanism, thus influencing which policy is chosen and applied, and which others are discarded.

# 3. POSSIBLE APPROACHES

In this section we explore some possible approaches, in particular we discuss about object-, ontology- and data-oriented approach. The reason of a such choice is that the first is a clear example of how the middleware itself can evolve, while the second is an example of how the middleware can support the adaptation of the applications that run over it. The last approach, instead, shows in which manner is possible to add context-awareness to existent middleware.

## 3.1 Object-oriented

We start discussing the use of object oriented reflection as a principle way to achieve a middleware that is adaptable and extensible, and thus capable of supporting context-aware applications.

### *3.1.1 Reflection in a nutshell*

Smith in [22] introduced the following reflection hypothesis:

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representation of that world, so too a computation process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*

This hypothesis implies that the system has some representation of itself in terms of programming structures available at run time. In a reflective system we call *base-level* the part of the system that perform processing about the application domain as in conventional systems (for a middleware it can be considered as the services it provides through its interfaces), *meta-level* the part of the system whose subject of computation is the system's self representation, and meta-object the entities that populate the meta-level. The process through which the self-representation is created is known as *reification*. A single meta-object is the result of a process of reification of some aspects of the system.

### *3.1.2 General principles*

The reflective architecture that we discuss was proposed in [6] in which both base- and meta-level are modelled and programmed according to an uniform object model whose design is based on the following principles. First, the procedural reflection is adopted because this approach has several advantages over a more declarative approach [17]. Second, RM-ODP Computational Model [1] is adopted and the computational interfaces are described using CORBA-IDL. The main features of this model are: *i)* objects can have multiple interfaces, *ii)* operational (based on method invocation semantics), stream (interactions by means of continuous flows of data) and signal (support to one-way interactions) interfaces are supported, *iii)* explicit bindings (local or distributed) can be created between compatible interfaces. Third, in order to restrict the effects of reflective computation to the reified objects, avoiding side-effects on other parts of the platform, there is an association of meta-objects with individual base-level objects of the platform. Finally, the meta-space is structured as a number of closely related but distinct meta-space models. The main feature of this choice is to simplify the interface offered by meta-space by maintaining a separation of concern between different system aspects.

### *3.1.3 The structure of meta-space*

As discussed above, the meta-level is divided into independent and orthogonal meta-models (as shown in fig. 4), each one representing a different aspect of the platform.

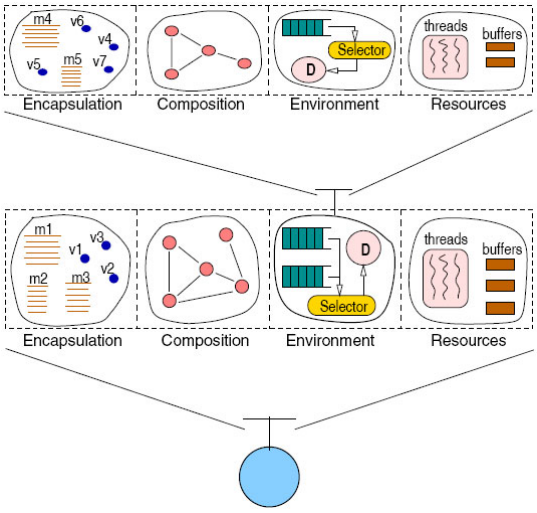


**Figure 4: Overall structure of meta-space.**

*Encapsulation.* The encapsulation meta-model relates to the set of methods and associated attributes of a particular object interface. It provides operations to add or remove methods and attributes.

*Compositional.* The compositional meta-model deals with the way a composite object is composed, i.e. how its components are inter-connected, and how these connection are manipulated. The composition of an object is represented as an object graph, in which the constituent objects are connected together by edges representing local bindings. Note that some objects in this graph can also be distributed. In practice, the composition meta-model provides operation to inspect and adapt the composition object, allowing view the structure of the graph, access individual objects and insert or remove component.

*Environment.* The environment meta-object is in charge of the environmental computation of an object interface, i.e.

how the computation is done. It deals with message arrivals, dispatching, marshalling, concurrency control, etc. Different levels of access are supported. For example, a simple meta-object may only deal with the arrival and dispatching of message at the particular interface. A more complex meta-object can add additional levels of transparency or control over thread creation and scheduling. Note that the environment meta-model it's represented as a composite object and thus it's possible to inspect and adapt it at the meta-meta-level using compositional meta-model.

*Resource Management.* The resource meta-model is concerned with both the resource awareness and resource management of objects in the platform. Resource provide an operating system independent view of threads, buffer, etc. Resource are managed by resource managers, which map higher level abstraction of resource onto lower ones. A complete description of the Resource Management Framework can be found in [8].

### 3.1.4 Example

In this part an example is presented to illustrate the use of the meta-models presented above to dynamically adapt the services provided by the middleware. More precisely the following example is concerned about the use of the compositional meta-model, but other examples can be found in [6][5].

Figure 5 shows a two-levels binding between the interfaces of two application objects that may be used, for example, to transfer a continuous stream of media from one object to the other with some constraint of QoS. At run-time, some external monitoring mechanism notices a drop in the network throughput that might lead to the violation of constraints. To avoid this we can reduce the actual amount of data to be transfered inserting compression and decompression filters at both sides of the binding.

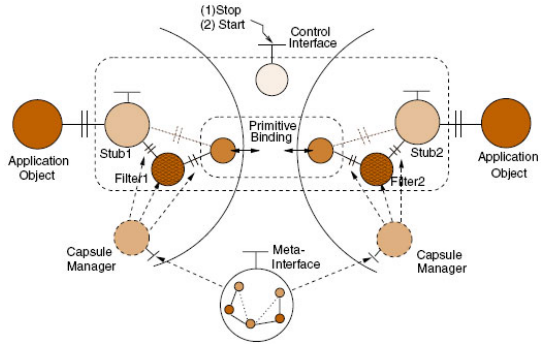


**Figure 5: Adaptation using the compositional meta-model.**

As the picture shows, the compositional meta-object (`meta-obj`) maintains a representation of the binding configuration (the object graph). The QoS monitor can invoke the operations provided by the compositional meta-model and any results are reflected in the actual configuration of components in the binding object. In this particular case, an object named Filter1 (respectively Filter2) is inserted between the interfaces of the Stub1 (respectively Stub2) and the primitive binding. To operate this reconfiguration, the following two calls are made to the meta-object:

```
meta_obj.addComponent(filter1,
            (stub1.interf2, prim_binding.interf1))
meta_obj.addComponent(filter2,
            (stub2.interf2, prim_binding.interf1))
```

## 3.2 Ontology-oriented

In this section, we discuss a Service-Oriented Context-Aware Middleware (SOCAM) architecture [11] for building of context-aware services based on a formal context model based on ontology using OWL [23] to address issues including semantic context representation, context reasoning and knowledge sharing, context classification.

### 3.2.1 Advantages of an ontology-based approach

We start discussing the benefits that this kind of approach involves. First of all, using an ontology-based approach allows us to describe contexts semantically in a way which is independent of the programming language or underlying operating system. Moreover it enables formal analysis of domain knowledge. For example, context reasoning becomes possible through first-order logic. There is another important aspect that is made possible to achieve through the use of an ontology: the sharing of common understanding of the structure of context information. In other word, an appropriate context model should enables the common schemas to be shared between different entities.

### 3.2.2 Design of context ontology

During the design of an appropriate context ontology several features of the context information must be taken in account: *i)* context has a great variety, i.e. context can describe any information in any domain; *ii)* context information varies in different sub-domains, e.g. we are more interested about devices like a TV and DVD player in an home domain rather than a PC in an office domain; *iii)* context information is interrelated, e.g. the current status of a person (Showering) is closely related to his location (Bathroom), the status of the water heater (On) and the shower door's status (Closed); *iv)* context information is inconsistent, e.g. the bedroom location sensor may sense a person is not present in his bedroom whereas the camera senses his presence.
As discussed above, context has a great variety and thus it's unrealistic to be processed efficiently by pervasive devices which have limited resource as CPU speed and memory. In consequence of this observation a two-layer hierarchical approach was adopted for designing the context ontology as shown in figure 6.

The context ontology is divided into common upper ontology and several domain-specific ontologies. The former captures general concepts about the physical world in pervasive computing environments, is fixed once defined and will be shared among different domains. The domain-specific ontologies are a collection of low-level ontologies which define the details of general concepts and their properties in each sub-domain. A low-level ontology can be dynamically plugged into and unplugged from the upper ontology when the environment is changed. The separation of domain reduce the burden of context processing and make it possible to reason about context knowledge on pervasive devices.

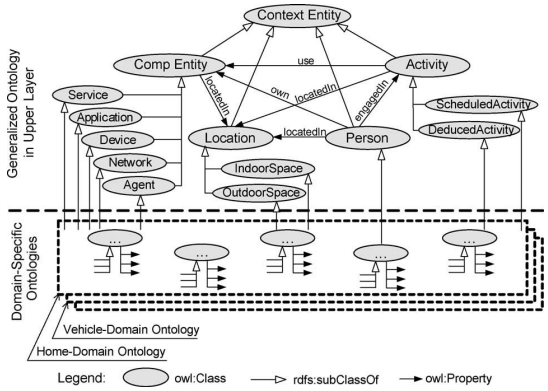


**Figure 6: Class hierarchy of the upper ontology.**

### 3.2.3 Architecture

The SOCAM architecture (fig. 7), based on the context model presented above, provide an efficient infrastructure for building context-aware services in pervasive computing environments. Note that is a clear example of how the middleware itself can be distributed, indeed it consists of several components which act as independent service components.

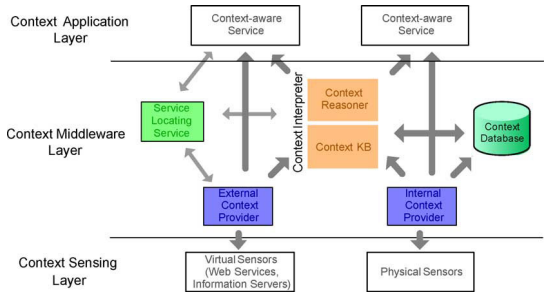


**Figure 7: Overview of the SOCAM architecture.**

*Context Providers.* Context Providers abstract useful context information from different sources and convert them to OWL representation so that knowledge can be shared and reused by other SOCAM components. The sources can be external or internal. The external providers obtain context information from external sources, e.g. a weather information server, whereas the internal providers acquire contexts directly from ubiquitous sensors located in a sub-domain. In order to be discovered by other components, each context provider needs to be registered into a service registry by using the Service Locating Service mechanism.

*Context Interpreter.* Context Interpreter consists of Context Reasoning Engines and Context KB (Knowledge Base). The context reasoner provides deduced contexts based on direct contexts and detect inconsistency in the context KB. Different inference rules can be specified and loaded into the reasoning engines. The context KB provides services that other components can invoke to query, add, delete or modify context knowledge stored in the Context Database. In SOCAM, two kinds of reasoning are currently supported: ontology reasoning (includes RDFS reasoning and OWL reasoning) and user-defined rule-based reasoning (provides forward chaining, based on the standard RETE algorithm, backward chaining rule engine, that uses a logic programming engine similar to Prolog engines, and a hybrid execution model, that performs reasoning by combining both forward-chaining and backward-chaining engines).

*Context-aware Services.* Context-aware Services adapt the way they behave according to the current context. To obtain contexts, a context-aware service can either query a context provider (by querying the service registry provided by the service locating service) or listen for events sent by context providers.

*Service locating service.* Service Locating Service provides a mechanism through which users or applications can locate and access to Context Providers and Context Interpreter. The service locating service mechanism is widely discussed in [24].

### 3.2.4 Performance evaluation

SOCAM middleware has been implemented in Java using J2SE 1.3.1 and a prototype in a smart home environment has been developed which consists of an OSGi-compliant residential gateway (OSGI). The gateway was designed based on Intel Celeron 600 MHz CPU with 256 MB memory. It runs embedded Linux (kernel 2.4.17) operating system and supports various wired and wireless network connections so that various devices such as PCs, PDAs and network cameras can be connected to the gateway. The context interpreter has been implemented using Jena2-HP's Semantic Web Toolkit. The domain-specific ontologies in both home and vehicle have been developed in OWL. The home-domain ontology consists of 89 classes and 156 properties and the vehicle-domain ontology consists of 32 classes and 57 properties. Based on the SOCAM middleware, a number of context-aware services in a smart home environment have been developed such as context-aware services for smart phone, home energy saving services, happy dining room services.

The context interpreter, which ran on the residential gateway, loaded both the upper ontology and a domain-specific ontology and merged them together. The operations of loading and merging involve checking the ontologies for inconsistencies. As figure 8 shows, the overhead (the time for loading and merging the upper ontology) is low and it can be further reduced when a domain-specific ontology is extended.

In the prototype, the context interpreter takes about 521 ms to load 96 context instances from various internal context providers and takes about 20 ms to merge these instances with the ontology. The context reasoning process takes about 1.9 s to derive high-level contexts. The context interpreter was able to answer queries for derived contexts at the average rate of a few milliseconds per query. The result shows that the logic reasoning is a computationally intensive process and it may become the bottleneck when it is applied to pervasive computing domain. Figure 9 shows the reasoning performance over difference scales of context knowledge. The reasoning performance can be increased by means of user-defined rule-based reasoning [11].

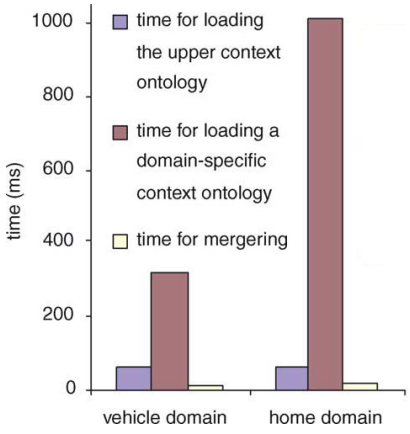


**Figure 8: Overhead of the two-layer ontology design.**

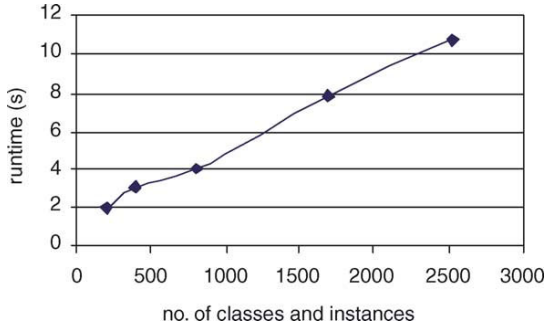


**Figure 9: The reasoning performance.**

## 3.3 Data oriented

For the "Data oriented" approach, we have studied two context-aware middleware. The first approach is based on the Publish-Subscribe with the add of context, created by "Politecnico di Milano" [7]. The second approach is about tuple-space, Tuples On The Air (TOTA), created by "Università di Modena e Reggio Emilia" [18].

### *3.3.1 Context-Aware Publish-Subscribe*

In most domains in which content-based publish-subscribe finds its natural application, an effective communication paradigm requires to take into account the situation in which the information to be communicated is produced or consumed. So, not only the informative content of messages is relevant to determine the information flow, but also the context in which this information has been produced and its relationship with the context of the consumer. This need for context-awareness is common in publish-subscribe application. To convey this information into the published messages, the publisher encoded the context into the messages. This approach is limiting and inefficient in routing messages from publisher to subscribers.

Several publish-subscribe applications have a common characteristic: both the subscribing and publishing of messages are tied to the context in which producing and consuming entities are. So, we want a new communication paradigm that explicitly takes context into account.

Shortly, we examine some properties about why we need a new communication paradigm on publish-subscribe:

- **Matching inversion**: In conventional content-based publish-subscribe systems messages hold data, while subscriptions hold constraints on these data. For the publish-subscribe considered, we need to invert the conventional matching process to consider constraints embedded into messages and data embedded into subscriptions.

- **Efficiency**: Managing context explicitly enables a dispatching strategy that limit the spreading of subscriptions only to those areas of the routing network matching publishers exists. This reduce the overhead of the subscriptions and unsubscription processes and reduce the time required to match messages.

- **Separation of concerns**: Usually the components in charge of publishing messages and subscribing to them differ from those in charge of detecting and communicating context changes. Tying the two concept together might reduce the readability of code, forcing interaction among parts of the application that should be kept separate.

**API**

To get over the limitation of content publish-subscribe has been proposed the following API. This interface introduce *context* as a first class element into the publish-subscribe. Each node $n$ can set its current context by invoking the *setContext(c)* operation and subscribe to messages matching the *content filter* $f_{msg}$ and coming from publishers whose context matches the *context filter* $f_{ctx}$ through the *subscribe*($f_{msg}$; $f_{ctx}$); the *unsubscribe*($f_{msg}$; $f_{ctx}$) operation does the opposite. Additionally, the node n can publish messages for subscribers whose context matches the context filter $f_{ctx}$ by invoking the **publish(m;** $f_{ctx}$**)** operation.

**Shortest Path Context Forwarding protocol**

To support large scale scenarios that involve hundreds of nodes, the *"Shortest Path Context Forwarding" (SPCF)* protocol has been developed. The SPCF protocol define how a set of brokers connected in an overlay network should cooperate in order to efficiently provide the context-aware publish-subscribe service to their clients. Messages are forwarded along the shortest path tree rooted at the publisher, using information about context and interests of downstream clients to decide the branches to follow and those to prune.

Each broker runs a link state protocol to built its own view of the dispatching network (without the client) and calculate the *shortest path trees (SPT)* (i.e. Dijkstra) rooted to each broker in the network. Then, message forwarding uses these trees together with two tables (see figure 10):

- **context table:** it maps brokers (identifier) to the set of contexts of their clients;

- **content table:** it stores, for each broker $B_p$, each context $c_p$ among those of the clients attached to $B_p$, and

each neighbor $N$, the set of content filters and contexts coming from attached to brokers that are downstream along $N$ in the $SPT$ rooted at $B_p$.

| Context table | |
|---|---|
| broker id | $\{c_1, ..., c_n\}$ |

| Content table | | | |
|---|---|---|---|
| broker id | context | neighbor id | $\{(f_{mag_1}, c_1), ..., (f_{mag_n}, c_n)\}$ |

**Figure 10: Tables kept by each broker.**

### 3.3.2 Tuples On The Air (TOTA)

*"Tuples On The Air" (TOTA)* is a novel middleware for supporting adaptive context-aware application in dynamic network. The key objectives of TOTA are:

- to promote uncoupled and adaptive interactions by locally providing application components with simple, yet highly expressive, contextual information;
- to actively support adaptively by discharging application components from the duty of dealing with network and application dynamics.

To this end, TOTA relies on spatially distributed tuples, to be injected in the network and propagated accordingly to application-specific patterns. On the one hand, tuple propagation patterns are dynamically re-shaped by the TOTA middleware to implicitly reflect network and applications dynamics, as well as to reflect the evolution of coordination activities. On the other hand, application components have simply to locally "sense" tuples to acquire contextual information, to exchange information with each other, and to implicitly and adaptively orchestrate their coordination activities.

TOTA is composed by a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. Each TOTA node holds references to a limited set of neighboring nodes. The structure of the network, as determine by the neighborhood relations, is automatically maintained and updated by the node to support dynamic changes.

**TOTA tuples**

In TOTA, distributed tuples used for both representing contextual information and enabling uncoupled interaction among distributed application components. Unlike traditional shared data space models, tuples are not associated to a specific node of the network but they are injected in the network accordingly to a specific pattern. TOTA tuples are able to express not only messages to be transmitted between application components but, more generally, some contextual information on the distributed environment.

TOTA tuples are injected into the system from a particular node, and spread hop-by-hop accordingly to their propagation rule. A TOTA tuple is define in terms of a "content" and a "propagation rule" $T = (C, P)$. The *content* $C$ is an ordered set of typed fields representing the information carried on by the tuple. The *propagation rule* $P$ determines how tuple is distributed and propagated in the network. This includes determining the "scope" of the tuple and how propagation can be effected by the presence or the absence of other tuples in the system. Tuples are not necessarily distributed replicas, so propagation rule can determine how tuple's content should change while is propagated.

**Architecture**

The TOTA middleware is constituted by three main parts:

- **TOTA API:** It is the main interface between the application and the middleware. It provide functionalities to let the applications inject new tuples in the system, to access the local tuple-space or to place subscriptions in the event interface.
- **Event interface:** It is the component in charge of asynchronously notifying the application about the income of new tuple or about the fact a new node has been connected or disconnected to the node's neighborhood.
- **TOTA engine:** It is the core of TOTA; it is in charge of maintaining the TOTA network by storing the references to neighboring nodes and to manage tuple's propagation by opening communication socket to send and receive tuples.

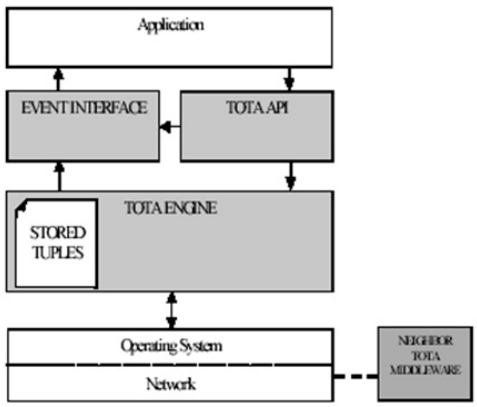


**Figure 11: The architecture of TOTA**

TOTA is provided with a simple set of primitive operations to interact with the middleware. The whole operation are:

- **void inject(Tuple t)**: it is used to inject the tuple passed as an argument in the TOTA network;
- **ArrayList read(Tuple t)**: it returns a collection of tuples locally present in the tuple space and matching the template passed as parameter;
- **ArrayList delete(Tuple t)**: it extract from the local middleware all the tuples matching the template and return them to the invoking component;
- **void subscribe(Tuple t, String reaction)**: it associates the execution of a reaction method in the component in response to the occurrence of events matching the template tuple passed as first parameter;

- **void unsubscribe(Tuple t, String reaction)**: it removes matching subscriptions.

These primitives rely on the fact that any event occurring in TOTA can be represented as a tuple.

# 4. SUMMARY

We have discussed why the construction of context-aware applications is difficult and indicated the support that software engineers can expect from current middleware products to simplify this task. We defined the terms context and context-awareness and than surveyed the literature in this area, discussing current approaches to sense and model the context.

# 5. REFERENCES


[1] G. S. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29:929–945, 2003.

[3] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[4] H. Chen, T. Finin, and A. Joshi. Using owl in a pervasive computing broker, 2003.

[5] F. M. Costa, G. S. Blair, and G. Coulson. Experiments with reflective middleware. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98 Workshop Reader*. Springer-Verlag, 1998.

[6] F. M. Costa, H. A. Duran, N. Parlavantzas, K. B. Saikoski, G. S. Blair, and G. Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In *OORaSE*, pages 79–98, 1999.

[7] G. Cugola, A. Margara, and M.Migliavacca. Context-aware publish-subscibe: model, implementation and evaluation.

[8] H. A. Duran and G. S. Blair. A resource management framework for adaptive middleware, 2000.

[9] F. Eliassen, A. Andersen, G. S. Blair, F. Costa, G. Coulson, V. Goebel, Øivind Hansen, T. Kristensen, T. Plagemann, H. O. Rafaelsen, K. B. Saikoski, and W. Yu. Next generation middleware: Requirements, architecture, and prototypes. In *In Proceedings of the 7 th IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 60–65. IEEE Computer Society Press, 1999.

[10] T. Gu, H. K. Pung, and D. Q. Zhang. A middleware for building context-aware mobile services. In *In Proceedings of IEEE Vehicular Technology Conference (VTC*, 2004.

[11] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28:1–18, 2005.

[12] S. Hadim, J. Al-Jaroodi, and N. Mohamed. Trends in middleware for mobile ad hoc networks. *JCM*, 1(4):11–21, 2006.

[13] B. Hardian. Middleware support for transparency and user control in context-aware systems. In *MDS '06: Proceedings of the 3rd international Middleware doctoral symposium*, New York, NY, USA, 2006. ACM Press.

[14] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam. Middleware for distributed context-aware systems. In *International Symposium on Distributed Objects and Applications (DOA*, pages 846–863. Springer, 2005.

[15] N. Hung, N. Ngoc, L. Hung, S. Lei, and S. Lee. A survey on middleware for context-awareness in ubiquitous computing environments.

[16] D. S. Linthicum. *B2B application integration: e-Business—enable your enterprise*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2000.

[17] P. Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[18] M. Mamei, F. Zambonelli, and L. Leonardi. Tuples on the air: a middleware for context-aware computing in dynamic networks, 2003.

[19] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15:279–328, 2006.

[20] H. Q. Ngo, A. Shehzad, S. Liaquat, M. Riaz, and S. Lee. Developing context-aware ubiquitous computing systems with a unified middleware framework. pages 672–681. 2004.

[21] A. Ranganathan and R. H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 143–161, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[22] B. C. Smith. *Procedural reflection in programming languages*. PhD thesis, MIT, Cambridge, Mass., 1982. Available as MIT Laboratory of Computer Science Technical Report 272.

[23] M. K. Smith, C. Welty, and D. L. McGuinnes. Web Ontology Language (OWL) guide, August 2003.

[24] T. G. Xiao, H. C. Qian, J. K. Yao, and H. K. Pung. An architecture for flexible service discovery in octopus. In *Computer Communications and Networks, 2003. ICCCN 2003. Proceedings. The 12th International Conference on*, pages 291–296, Oct. 2003.

[25] T. G. Xiao, X. H. Wang, H. K. Pung, and D. Q. Zhang. An ontology-based context model in intelligent environments. In *In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 270–275, 2004.