Middleware for Ambient Intelligence Environments: Reviewing Requirements and Communication Technologies

Yannis Georgalis[1], Dimitris Grammenos[1], and Constantine Stephanidis[1,2]

[1] Institute of Computer Science, Foundation for Research and Technology – Hellas (FORTH), GR-70013 Heraklion, Crete, Greece
[2] Computer Science Department, University of Crete, Greece
{jgeorgal,gramenos,cs}@ics.forth.gr

Abstract. Ambient Intelligence is an emerging research field that aims to make many of the everyday activities of people easier and more efficient. This new paradigm gives rise to opportunities for novel, more efficient interactions with computing systems. At a technical level, the vision of Ambient Intelligence is realized by the seamless confluence of diverse computing platforms. In this context, a software framework (middleware) is essential to enable heterogeneous computing systems to interoperate. In this paper we first consider the basic requirements of a middleware that can effectively support the construction of Ambient Intelligence environments. Subsequently, we present a brief survey of existing, general-purpose middleware systems and evaluate them in terms of their suitability for serving as the low-level communication platform of an Ambient Intelligence middleware. Finally, we argue that an Object-Oriented middleware such as the Common Request Broker Architecture (CORBA) is most suited for basing a middleware for Ambient Intelligence environments.

1 Introduction

1.1 Ambient Intelligence

The term Ambient Intelligence (AmI) describes those environments that enclose a plethora of diverse computing systems, embedded to, and indistinguishable from, the environment in which they operate [1]. An AmI infrastructure aims to support users in carrying out their everyday life activities by offering them an easy and natural way for interacting with the digital services that are provided by the hidden interconnected computing systems. In this respect, AmI environments provide the means to sense and construe the actions that serve the needs of their users in order to offer a personalized, context-sensitive and efficient interaction platform.

1.2 Distributed Services

In an environment where interactions are realized by the confluence of different interconnected computing systems, the organization of the overall system architecture to a well-defined set of distributed software entities is crucial. The alternative centralized approach where all software entities run on a monolithic computing platform, despite being the easiest to implement, is neither scalable nor flexible. On the contrary, a distributed approach allows for (a) flexible, dynamic extension of the overall system with novel functionality, (b) enhanced system scalability, by sharing computation demands among different computers, (c) enhanced robustness, by isolating potential failures of individual software entities, and (d) unambiguous and straightforward modularization of the system's architecture.

Therefore, we consider an AmI infrastructure as a collection of interconnected distributed services; i.e., a collection of software entities that run on different machines, and are able to communicate with each other in order to provide to the infrastructure all the required functionality for sensing, drawing conclusions, and responding to the needs of its users.

2 Basic Requirements for an Ambient Intelligence Middleware

2.1 Role of the Middleware

Despite being an overloaded term [2], middleware is a commonly used word in the context of distributed computing systems. In general, middleware is a set of programming libraries and programs (services) that constitute an indivisible platform which offers a comprehensible abstraction over the complexities and potential heterogeneity of the target problem domain.

Different communication middleware platforms support different programming and communication models. Three of the most popular paradigms are the object-based middleware, the event-based middleware and the message oriented middleware (MOM). In object-based middleware platforms, applications are structured into distributed objects that interact via location-transparent method invocation. Those platforms typically utilize the "request/response" communication style. On the other hand, event- and message- based systems mainly employ single-shot message exchange. Event-based middleware is particularly suited to the construction of non-centralized distributed applications that ultimately monitor and react to changes in their environment.

In an inherently distributed environment such as AmI, the communication middleware should abstract over the intricacies of the underlying communication technologies, machine architectures and operating systems. Moreover, it should hide the distribution of the different parts that comprise the system and enable programs written in different programming languages to communicate seamlessly. Higher-level core functionality of an AmI infrastructure, e.g. context awareness, authentication, etc., basically comprises a set of appropriate services that depend on the middleware.

2.2 AmI Middleware Basic Requirements

Enabling programs written in different programming languages to interoperate seamlessly is a key design goal in most middleware platforms. This is especially true in an AmI environment, where the basic system is built using diverse technologies from diverse research fields that traditionally utilize different programming languages.

Synchronous communication is definitely essential for interacting with different AmI services. Using synchronous calls a service or an application can give commands to other services and query their internal state. However, the synchronous paradigm alone is inadequate for modeling all the interactions that can happen in an AmI environment. For this reason, asynchronous, event-based communication support is also needed in order to enable AmI services to notify interested parties about changes in their internal state, or to communicate the occurrence of an external (expected or unexpected) event. Event-based communication is also traditionally regarded as a mechanism for constructing loosely-coupled software components that need only know the format of the exchanged events without requiring any knowledge of the internal structure, implementation, or semantics of the entity that produced the event. Consequently, we consider both synchronous and asynchronous communication styles essential for an AmI middleware platform.

Arguably, one of the most important and extensively researched properties of distributed systems is fault tolerance. Fault tolerance, in this context, refers to the property that enables an AmI infrastructure to continue to function properly even in the event of failures. Apparently, the failure of an AmI service that serves a specific function within an infrastructure should not, in any case, affect the other services that do not depend or use this failing service. On the other hand, services and applications that depend on a failing service, should handle gracefully its failure by continuing functioning with (potentially) reduced functionality. Ensuring that a failing service will not affect independent services is definitely the job of the middleware. However, the graceful handling of the failure of a dependent service cannot entirely be handled at the middleware level. The failback techniques ultimately depend on the semantics of the high-level task at hand. Therefore, we regard fault tolerance, in the context of an AmI communication middleware, as the set of functionality that supports the following: (a) isolates failures, (b) eliminates single points of failure within the core middleware infrastructure, (c) is able to restart failing services before the clients that use those services are affected, and (d) provides mechanisms for notifying higher level entities about the irreparable failure of a specific service.

Another important requirement of an AmI middleware is security. Apparently, security, in order to be effective, should be considered throughout all the layers of an AmI infrastructure. In this context, however, we view security as the ability of the middleware to prevent malicious code from eavesdropping and exploiting the data exchanged through the network channels that enable services to communicate with each other.

Orthogonally to the aforementioned requirements, a key property of the middleware that we consider absolutely essential is that it should be easy for developers to program with. Developing AmI components should feel natural to the programmer of all supported languages, who should be able to use distributed services as if they were local objects or functions of the source language. Going one step further, the middleware should limit, or even eliminate entirely, "boilerplate" (i.e. extraneous) code in order to enable the construction and usage of distributed services.

An overview of the aforementioned requirements for an AmI middleware is presented in Table 1.

Table 1. Basic requirements of an AmI middleware

Heterogeneity	Support for multiple languages and computing platforms
Communication	Synchronous and asynchronous (event-based) communication
Resilience	Replication; isolation and graceful handling of failures
Security	Secure communication among distributed services
Ease of use	Natural and intuitive usage for each target language

3 Communication Technologies

Implementing a middleware that is able to satisfy all the aforementioned requirements requires substantial development effort. Fortunately, many existing communication technologies can be re-used towards this goal.

In the following subsections, we will present the primary communication technologies that we have evaluated with respect to their appropriateness as the basis for a communication platform for an AmI middleware.

3.1 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) [3] is a standard defined by the Object Management Group (OMG) [4] that provides a stable model for distributed object-oriented systems. CORBA enables software components written in multiple programming languages and running on multiple operating systems to work seamlessly together. The abstractions provided by CORBA Object Request Brokers (ORBs) allow for the creation and usage of distributed objects that look like typical local objects of the target programming language. The CORBA standard also defines a plethora of standard services that can be used to make the development of distributed systems easier and more robust.

There are many advantages in using CORBA as a base for constructing a middleware for AmI environments. First of all, CORBA separates the definition of interfaces from their actual implementation using an interface definition language (IDL). The standard specifies a "mapping" from IDL definitions to specific programming constructs of the target implementation language. This mapping process enables the type safe invocation of the methods offered by distributed services, simplifies their implementation, and provides a comprehensive formal reference of the Application Programming Interface (API) that a specific service supports.

CORBA is primarily designed for blocking request/response, synchronous type of communication. However, using the standard Notification Service [5], CORBA conformant applications can use publish/subscribe channels to effectively emulate asynchronous communication. Additionally, support for one-way method invocation, allows callers to continue execution without waiting for any response from the server.

Fault tolerance in CORBA is not specifically addressed in earlier revisions of the standard. However, it allows for a high degree of fault tolerance by: (a) having clear invocation failure models (at-most-once and best effort delivery); (b) allowing clients to obtain persistent references to services through the standard Implementation Repository service [5]; (c) enabling object references to include multiple endpoints. Additionally, CORBA allows client code to register functions to intercept exchanged messages, enabling the creation of more advanced fault tolerance methods, depending on the target problem domain.

Concerning the "ease of use" requirement, CORBA is arguably difficult to use. Nonetheless, having a versatile architecture, it allows for the construction of higher level communication platforms on top of it. Despite having many obscure, esoteric and obsolete features, a higher-level platform for AmI environments can definitely use a well-defined subset of CORBA and abstract away its "idiosyncrasies".

Furthermore, there are many high quality open source CORBA implementations for many different programming languages (e.g. [6], [7], [8], [9]). All these implementations have liberal licenses allowing client applications to be distributed at their authors' own terms. Table 2 summarizes the key features of CORBA against the five requirements set in section 2.2.

Table 2. Summary of CORBA features

Heterogeneity	Supports multiple programming languages and computing platforms
Communication	Synchronous request/response and asynchronous communication through the standard Notification Service
Resilience	Comprehensive invocation failure semantics, mechanisms for supporting transparent replication
Security	Ability to use encrypted communication channels
Ease of use	Natural and intuitive usage for each target language

3.2 Internet Communications Engine (Ice)

The Internet Communications Engine (Ice) [10] is a true object-based domain-independent middleware platform designed and implemented by ZeroC [11]. Ice derives its main architecture from CORBA, but tries to improve on it by (a) eliminating its unnecessary complexity, (b) providing better built-in security, (c) providing more efficient protocols for reduced network bandwidth and CPU overhead, and (d) providing extra functionality that is either underspecified or absent from the CORBA standard and its implementations.

Slice, Ice's equivalent of CORBA's IDL, extends the latter by adding functionality for supporting dictionary types and by providing support for exception inheritance. Additionally, Slice allows the programmer to add directives describing the state of Ice objects so that they can be subsequently stored and loaded automatically.

Ice offers many standard services including a service for propagating software updates around the distributed infrastructure (IcePatch), a very efficient Notification service (IceStorm), and a transparent proxy server that can be used for firewall traversal and enhanced security (Glacier).

All in all, Ice succeeds in delivering a well-designed middleware that without trying to reinvent the wheel, offers a robust and easy to use platform for distributed computing[1]. That said, Ice's disadvantages compared to CORBA implementations are stemming from purely practical reasons. On one hand, Ice's GPL [12] license requires all the applications and services that use its libraries and generated code to be

[1] It is worth noting that ZeroC comprises former CORBA implementers and a member of OMG's Architecture Board (Michi Henning).

distributed under the GPL – a requirement that we have considered as too restrictive. On the other hand, the extra features offered by Ice were not deemed essential for the implementation of an AmI middleware. Nonetheless, ZeroC, in addition to the GPL license, offers proprietary licensing schemes for a fee. Consequently, should a proprietary licensing scheme be a viable option, Ice is an ideal communication platform for basing an AmI middleware. Table 3 summarizes the key features of Ice against the five requirements set in section 2.2.

Table 3. Summary of Ice's features

Heterogeneity	Supports multiple programming languages and computing platforms
Communication	Synchronous request/response and asynchronous communication through the IceStorm service
Resilience	Comprehensive invocation failure semantics, mechanisms for supporting transparent replication
Security	Ability to use encrypted communication channels, Glacier service
Ease of use	Natural and intuitive usage for each target language

3.3 Web Services

Web services [13], being the new Internet standard for service provision, are widely used in modern distributed systems. This technology uses a simple XML-based protocol to allow applications to exchange data across the Web. Services themselves are defined in terms of the well-defined XML documents – modeling messages – that are accepted and generated. Instead of providing a specification that offers high-level, standardized language-specific constructs for mapping service interfaces and data types, web service aware code, only needs to be able to generate and process the exchanged XML documents. The Simple Object Access Protocol (SOAP), that essentially constitutes the core of the Web services architecture, only defines the format of the exchanged messages, the marshaling rules for the data that appear in the messages, and a set of conventions for achieving Remote Procedure Call- (RPC) like functionality.

Putting aside any performance considerations, especially in the context of contemporary high-speed networks, we found web services to be insufficient as a platform to base an AmI middleware. The potential advantages offered by an approach based on Web services such as universal firewall traversal, loose coupling of services and dynamic service composition, are outweighed by the disadvantages stemming from the absence of high-level programming idioms and communication guarantees in the specification.

In an object-based middleware, the implementation of a service is realized (in object oriented languages) as the implementation of a class. Similarly, a remote call to a service is realized as a method invocation on a local object that acts as a proxy to the remote service. To the programmer, a service implementation or invocation is identical to the implementation and invocation of a local object of the target programming language. Additionally, remote invocations (at least in CORBA) have well defined failure semantics; at-most-once for blocking, synchronous calls and best effort delivery for one-way non-blocking calls. On the other hand, in Web services, the programmer has to implement the dispatching of the received messages to the appropriate functions of the target language in order to implement a service and explicitly construct and send a SOAP message in order to invoke a remote function. Moreover, the SOAP specification omits the definition of invocation failure semantics. The lack of natural programming abstractions is mitigated by the provision of additional libraries and tools. However, such tools are not standard and are available only for just a few programming languages.

While universal firewall traversal is very important for geographically distributed services, it is not essential in the context of an AmI environment where the majority of the deployed services are restricted within a Local Area Network (LAN). Apparently, firewall traversal is also possible in CORBA by forcing a standard port to the Object Request Broker (ORB) of those services that should be visible from systems outside the basic infrastructure LAN and subsequently opening this port in the firewall either manually or through Universal Plug and Play (UPnP) messages. Also, the dynamic composition and invocation of Web services that need not know a priori the functions that a specific service supports is also possible in CORBA through its standard Dynamic Invocation Interface (DII). Table 4 summarizes the key features of Web Services against the five requirements set in section 2.2.

Table 4. Summary of Web Services features

Heterogeneity	Support multiple programming languages and computing platforms
Communication	Synchronous request/response. Asynchronous communication can be achieved but is not standard
Resilience	Depends on the underlying communication protocol and does not provide any mechanisms for supporting the implementation of resilience
Security	Ability to use encrypted communication channels
Ease of use	Explicit message construction and dispatching

3.4 Thrift

Thrift [14], used extensively in facebook [15], is a communication platform that emphasizes simplicity and efficiency in the delivery and invocation of distributed services. Naturally, Thrift enables the creation and usage of distributed services in many different programming languages. Using an Interface Definition Language (IDL) much like CORBA's IDL, Thrift effectively separates the description of a service from its actual implementation while providing a natural object-oriented mapping for using and implementing services in the supported languages. One particularly useful feature that Thrift supports is the fine-grained service versioning. Thrift is able to distinguish and handle gracefully differences in the version of every field of complex data structures and every parameter in a service function.

While Thrift is very well suited to the particular problem domain for which it was developed it does not provide all the mechanics required for an AmI middleware. Most importantly, it lacks the ability to use service references as first-class values and does not implement core infrastructure services such as Naming and Notification. The absence of a Notification service makes it impossible for services to notify clients asynchronously[2] about the occurrence of an event. Overall, while Thrift supersedes CORBA in terms of simplicity, efficiency and interface versioning, it lacks CORBA's large feature-set, flexibility, maturity and robustness. Table 5 summarizes the key features of Thrift against the five requirements set in section 2.2.

Table 5. Summary of Thrift's features

Heterogeneity	Supports multiple programming languages
Communication	Synchronous request/response. Asynchronous communication can be achieved but is not standard
Resilience	Depends on the underlying communication protocol and does not provide any mechanisms for supporting the implementation of resilience
Security	Ability to use encrypted communication channels
Ease of use	Natural and intuitive usage for each target language

3.5 Etch

Etch [16], which was originally derived from work on the Cisco Unified Application Environment [17], is a cross-platform, language- and transport-independent framework for building and consuming network services. Etch implements a Network Service Description Language (NSDL) which separates the description of a service from its actual implementation in the target language. The processing of an NSDL service description, allows client code to implement and invoke a distributed service as if it were a local object of the target language. However, like Thrift, Etch is not a pure object-based middleware as it cannot use a service object as the return value or parameter of a method. Nevertheless, it offers support for two-way communication between a service and its clients and simplifies security management by enabling connection authorization directives to be specified in NDSL. By supporting two-way communication, Etch is able to support effectively synchronous request/response and asynchronous communication. As far as standardized services are concerned, Etch currently provides a Naming Service for discovering deployed services and a Router Service for fault-tolerance and load balancing.

The features that are planned for Etch provide all the functionality that we consider essential for the implementation of an AmI middleware. However, in its present release (version 1.0), Etch is incomplete. Although it offers most of the aforementioned functionality, it does so supporting only Java and .NET (C#) for implementing and consuming services. When its specification is fully implemented, offering the planned functionality for more programming languages, Etch will constitute an effective and efficient platform for an AmI communication middleware. Table 6 summarizes the key features of Etch against the five requirements set in section 2.2.

[2] Thrift's *async* keyword for qualifying a service's function is equivalent to the *oneway* qualifier in CORBA which essentially makes the call of the function non-blocking for the client code (i.e. fire-and-forget).

Table 6. Summary of Etch's features

Heterogeneity	Currently (version 1.0) supports only two languages with more to come in subsequent versions
Communication	Synchronous request/response; explicit support for asynchronous type-safe communication
Resilience	Provides a Router service that can be used for service replication
Security	Ability to use encrypted communication channels and also provides support for high-level authentication functions
Ease of use	Natural and intuitive usage for each target language

4 Related Efforts

The Amigo project [18] uses the OSGi framework [19] for implementing services in Java, and the .NET Web Services framework and tools for implementing services in .NET. Hydra [20] uses a Web Services-based approach with custom peer-to-peer (P2P) network technologies for creating and consuming services. The CHIL project [21] uses Smartspace Dataflow [22] and ChilFlow [23] for the integration of autonomic perceptual components and follows an agent-based approach for the implementation of high-level services using JADE [24]. Communication in JADE relies on Java Remote Method Invocation (RMI) for Java-based agents and on CORBA for agents running on different platforms. These efforts are still under development and support only a narrow range of platforms, as they target mainly Java and .NET-based AmI infrastructures.

5 Summary and Conclusions

In this paper, we presented the basic requirements for an AmI communication middleware. Against these requirements, we evaluated a set of general-purpose communication technologies. Among these communication technologies, we found the Common Object Request Broker Architecture (CORBA) and the Internet Communications Engine (Ice) to be the most effective in providing the low-level building blocks for implementing a middleware for AmI environments. Both CORBA and Ice, provide a robust specification that has a very broad range of features that essentially make them independent of the target problem domain. They are sufficiently low-level so that specialized, high-level interaction patterns can be realized, and sufficiently high-level so that the need for tedious communication management and marshaling operations is alleviated.

Despite the fact that a versatile object-based middleware constitutes an effective platform for an AmI communication middleware, it is apparent that request/response communication is not always suitable. Most importantly, it is neither efficient nor effective for streaming large amounts of continuous data, e.g. video or audio. Hence, one final issue to note is that an AmI middleware should also utilize a separate MOM communication platform (e.g., ChilFlow) for streaming data while maintaining an object-based core for creating and controlling data streams.

Acknowledgements. This work has been supported by the FORTH-ICS internal RTD programme "AmI: Ambient Intelligence Environments".

References

1. IST Advisory Group 2003. Ambient Intelligence: From Vision to Reality, ftp://ftp.cordis.lu/pub/ist/docs/istag-ist2003_consolidated_report.pdf
2. Network Working Group. Request for Comments 2768, http://www.ietf.org/rfc/rfc2768.txt
3. Object Management Group. The Common Object Request Broker: Architecture and Specification. Object Management Group, Framingham, Mass. (1998)
4. The Object Management Group (OMG), http://www.omg.org
5. Object Management Group. CORBAservices: Common Object Services Specification. Object Management Group, Framingham, Mass. (1997)
6. The ACE ORB (TAO), http://www.cs.wustl.edu/~schmidt/TAO.html
7. JacORB, http://www.jacorb.org
8. IIOP.NET, http://iiop-net.sourceforge.net
9. omniORB, http://omniorb.sourceforge.net
10. Henning, M.: A new approach to object-oriented middleware. IEEE Internet Computing 8, 66–75 (2004)
11. ZeroC, http://www.zeroc.com
12. GNU General Public License, http://www.gnu.org/copyleft/gpl.html
13. Simple Object Access Protocol (SOAP), http://www.w3.org/TR/soap
14. Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable Cross-Language Services Implementation
15. facebook, http://www.facebook.com
16. Etch, http://cwiki.apache.org/ETCH
17. Cisco Unified Application Environment, http://www.cisco.com/web/developer/cuae
18. The Amigo project, http://www.hitech-projects.com/euprojects/amigo
19. OSGi Alliance. OSGi Service Platform Core Specification Release 4, http://www.osgi.org
20. The Hydra project, http://www.hydramiddleware.eu
21. The CHIL project, http://chil.server.de
22. The NIST Smart Space Project, http://www.nist.gov/smartspace
23. The ChilFlow System, http://www.ipd.uka.de/CHIL/projects/chilflow.php
24. Bellifemine, F., Poggi, A., Rimassa, G.: JADE – A FIPA-compliant agent framework