A Perspective on the Future of Middleware-based Software Engineering

Valérie Issarny, Mauro Caporuscio, Nikolaos Georgantas

A Perspective on the Future of Middleware-based Software Engineering

Valerie Issarny, Mauro Caporuscio, Nikolaos Georgantas

Valerie Issarny got her PhD and “Habilitation à diriger des recherches” in computer science from the University of Rennes I, France, in 1991 and 1997 respectively. She currently holds a “Directeur de recherché” position at INRIA. Since 2002, she is the head of the INRIA ARLES research project-team at INRIA-Rocquencourt. Her research interests relate to distributed systems, software engineering, mobile wireless systems and middleware. She is chairing the executive committee of the AIR&D consortium on Ambient Intelligence Research and Development. Further information about Valerie's research interests and her publications can be obtained from http://www-rocq.inria.fr/arles/members/issarny.html.

Mauro Caporuscio received the Ph.D. in Computer Science from the University of L'Aquila, Italy. In 2002, he was a “Professional Research Assistant” in the Software Engineering Research Laboratory at the Department of Computer Science - University of Colorado at Boulder. He has currently a Post-Doc position at INRIA-Rocquencort within the ARLES Project and is a member of the Software Engineering and Architecture Group at the University of L'Aquila. His research interests mainly include Formal Methods for Software Validation, Software Engineering, Software Architecture Analysis, Distributed Component-Based Systems, Middleware, and Physical and Logical Mobility.

Nikolaos Georgantas received his Ph.D. in 2001 in Electrical and Computer Engineering from the National Technical University of Athens, Greece. He is currently senior research scientist of INRIA with the ARLES research group at INRIA-Rocquencourt, France. His research interests relate to distributed systems, middleware, ubiquitous computing systems and service \& network architectures for telecommunication systems. He currently works on ad hoc system interoperability in service-oriented pervasive computing environments based on semantic technologies, specifically leading the work package related to Ambient Intelligence middleware for the networked home in the EU IST Amigo project.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

A Perspective on the Future of Middleware-based Software Engineering

Valerie Issarny
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, France.
valerie.issarny@inria.fr

Mauro Caporuscio
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, France.
mauro.caporuscio@inria.fr

Nikolaos Georgantas
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, France.
nikolaos.georgantas@inria.fr

Abstract

Middleware is a software layer that stands between the networked operating system and the application and provides well known reusable solutions to frequently encountered problems like heterogeneity, interoperability, security, dependability. Further, with networks becoming increasingly pervasive, middleware appears as a major building block for the development of future software systems. Starting with the impact of pervasive networking on computing models, manifested by now common grid and ubiquitous computing, this paper surveys related challenges for the middleware and related impact on the software development. Indeed, future applications will need to cope with advanced non-functional properties such as context-awareness and mobility, for which adequate middleware support must be devised together with accompanying software development notations, methods and tools. This leads us to introduce our view on next generation middleware, considering both technological advances in the networking area but also the need for closer integration with software engineering best practices, to ultimately suggest middleware-based software processes.

1. Introduction

The main purpose of middleware is to overcome the heterogeneity of the distributed infrastructure. Middleware establishes a new software layer that homogenizes the infrastructure's diversities by means of a well-defined and structured distributed programming model (see Figure 1). In particular, middleware defines:

- An Interface Description Language (IDL) that is used for specifying data types and interfaces of networked software resources.
- A high-level addressing scheme based on the underlying network addressing scheme for locating resources.
- An interaction paradigm and semantics for achieving coordination.
- A transport/session protocol for achieving communication.
- A naming/discovery protocol, naming/description convention, registry structure, and matching relation for publishing and discovering the resources available in the given network.

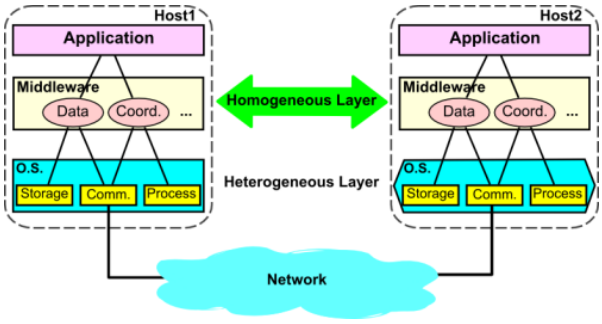

Figure 1. Middleware in Distributed Systems.

Over the years, the role of middleware has proven central to address the ever increasing complexity of distributed systems in a reusable way. Further, middleware provides building blocks to be exploited by applications for enforcing non-functional properties, such as dependability and performance. Attractive features of middleware have made it a powerful tool in the software system development practice. Hence, middleware is a key factor that has been and needs to be further taken into account in the Software Engineering (SE) discipline [22]. Methods and related tools are required for middleware-based software engineering. This need becomes even more demanding if we consider the diversity and scale of today's networking environments and application domains, which makes middleware and its association with applications highly complex.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

In this paper, we attempt a view on the present and future of middleware-based software engineering. As part of this view, we first look deeper into the role of middleware in building distributed systems, by surveying different middleware paradigms distinguished by the coordination model they offer to applications. We complement this overview with a first discussion of the impact of middleware on the software development process and the resulting requirements posed upon the latter (Section 2).

Even if legacy middleware has well been established and employed, the rapidly evolving computing and networking environments – including new devices and networks, new application domains, and complex associations among them – raise new, challenging requirements for middleware. We discuss this new context in which middleware has to operate and point out the resulting requirements (Section 3). Among those, access to computational resources should be open across network boundaries and dynamic due to the potential mobility of host- and user-nodes. This urges middleware to support methods and mechanisms for description, dynamic discovery and association, late binding, and loose coordination of resources. In such variable and unpredictable environments, operating not only according to explicit system inputs but also according to the context of system operation becomes of major importance, which should be enabled by the middleware.

In light of today's requirements for middleware, we present latest research efforts that attempt to deal with them and indicate still open issues. We thus sketch our view on the next generation of middleware-based software systems. We further discuss enabling software engineering techniques for these systems, and identify the needs for new methods and tools that will adequately support the advanced features of next generation middleware-based systems (Section 4). Thus, we first point out support for interoperability at both middleware and application level as key feature of future middleware-based systems; this is a reply to the high heterogeneity induced by ubiquitous computing as well as open and mobile networking. We further discuss open coordination in future systems as another feature enabling open and mobile networking. Then, we discuss the goal of dynamic adaptability to changing operating conditions, i.e., context, for next generation middleware-based systems. Further, based on the requirements for middleware-based software engineering support that we identify throughout this paper, we conclude Section 4 by outlining our view of a middleware-based software process, where the middleware has a central role in all phases of the software life-cycle. Finally, we close this paper by presenting our conclusions (Section 5).

2. The Role of Middleware

Abstractions provided by middleware systems hide the heterogeneity of the networking environment, support advanced coordination models among distributed entities and make as transparent as possible the distribution of computation. Management of distributed networked resources further goes with the provision of various non-functional properties related to dependability and performance management.

2.1. What is Middleware

Existing middleware platforms vary in terms of programming languages assumed for application development, and offered networking abstractions and related value-added services enforcing non-functional properties. Enhancing the taxonomy presented in [22], middleware platforms can be categorized according to the coordination model they implement, as surveyed below. Then, according to the specific model supported, development of middleware-based software systems may exploit dedicated or traditional Software Engineering (SE) methods and tools.

Transactional middleware

Transactions are contracts that guarantee a consistent system state transition and are used in various distributed application domains (e.g., applications centered on databases, telecommunications and safety critical systems). Technically, a transaction is a unit of coordination among subsystems, which in its most general form respects the ACID (Atomicity, Consistency, Isolation, and Durability) properties. In this setting, a transactional middleware offers an interface for running transactions among distributed components, while ACID properties may be possibly relaxed according to application requirements. Then, the SE research provides formal specification of transactional properties and offers tools for dealing with them [56].

Tuplespace-based middleware

The use of a coordination model centered on tuplespaces was initially introduced by the Linda language [27], building upon the distributed shared memory concept. Specifically, Linda introduces high-level abstractions over distributed shared memory, so as to ease management and thus reasoning about the distributed system state. In concrete terms, a tuplespace is globally shared among components, and can be accessed for inserting, reading, or withdrawing tuples. The important characteristic of the tuplespace model is that it introduces complete decoupling among components, in terms of both space and time. In fact, the components that coordinate using the tuplespace do need neither

to co-exist at the same time nor to have any explicit reference to each other. Hence, a tuplespace-based middleware offers an abstraction of the distributed tuplespace, which is further managed in order to assure non-functional properties like reliability, persistence and scalability. Then, development of distributed applications using such a middleware relies on associated programming language (or primitives). Reasoning about the software system behavior is further assisted through formal modeling of the language semantics [16].

Message-oriented middleware

Message-Oriented Middleware (MOM) can be seen as particular instance of tuplespace-based middleware where tuples are implemented as messages and the space is implemented by distributed message-queues. In this case, parties communicate with each other by publishing, selecting and reading queued messages. MOM can be further classified in two categories according to the message selection mechanism that is provided: (i) Queue-based middleware where messages are selected by means of queue membership, and (ii) Publish/Subscribe middleware where messages are selected by means of predicates. MOM provides functionality to publish, select and deliver messages with properties such as persistence, replication, real-time performance as well as scalability and security. Since MOM is a relatively young technology, the SE community is still investigating tools that assist thorough development of MOM-based applications [11].

Remote procedure calls

As a pioneering coordination paradigm for distributed computing, Remote Procedure Call (RPC) is a protocol that allows a component to invoke procedures executed on remote hosts without explicitly coding the details for these interactions. In this context, a RPC middleware offers services for (i) generating client/server stub, (ii) marshalling/unmarshalling data (e.g., input parameters and return values), (iii) establishing synchronous communication, as well as for (iv) assuring non-functional properties. Due to its obvious relation with conventional programming paradigms, well know standard SE techniques and methodologies can be applied for designing and analyzing RPC-based applications in order to assess the software quality.

Object and component oriented middleware

Just like procedural programming evolved towards object-oriented (OO) and further component-based (CB) programming, OO and CB middleware represent the natural evolutions of RPC-based middleware. Specifically, OO and CB middleware provide the proper abstractions for exploiting the respective programming paradigms in a fully distributed environment. In particular, OO (CB) middleware offers tools that allow engineers to (i) generate stubs from the object (component) interface specification, (ii) obtain the reference of the remote object (component) to interact with, (iii) establish synchronous communication, and (iv) invoke requested methods (operations) by marshalling and unmarshalling exchanged data. Both OO and CB middleware assure non-functional properties such as reliability, scalability, and security. Also, SE methods and related tools devised for OO and CB software, can be used for designing and developing high-quality distributed software applications [21, 51].

Service-oriented middleware

A further step in the evolution of component-based programming is towards the Service Oriented Architecture (SOA) paradigm that supports the development of distributed software systems in terms of loosely coupled networked services [41]. In SOA, networked resources are made available as autonomous software services that can be accessed without knowledge of their underlying technologies. Key feature of SOA is that services are independent entities, with well defined interfaces, which can be invoked in a standard way, without requiring the client to have knowledge about how the service actually performs its tasks.

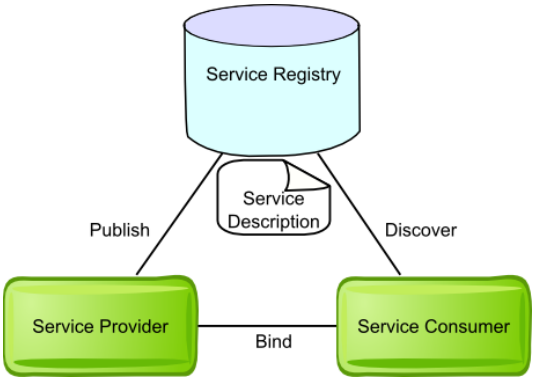

Figure 2. Service Oriented Architecture

The SOA style (see Figure 2) is structured around three key architectural components: (i) service provider, (ii) service consumer, and (iii) service registry. In SOA-based environments, the Service-Oriented Middleware (SOM) is in charge of enabling the deployment of services and coordination among the three key conceptual elements that characterize the SOA style. That is, SOM provides runtime support for service providers to deploy services on service host and further advertise their presence to the registry, and

for service consumers to discover and use services. The SOM hides heterogeneity of the underlying environment, by introducing languages for rigorous service description and protocols for service discovery and access. Although SOA has been a very active area of research over the last few years, actually taking over most middleware technologies, SOA and SOM are still relatively emerging technologies that are evolving at a high pace. Hence, SE research is investigating methods and tools that allows for the systematic development of service-oriented applications [19].

2.2. Middleware and Software Development

Middleware greatly facilitates the development of distributed applications, thanks to the definition of networking and computing abstractions that match distributed application requirements. As such, middleware becomes a key software engineering tool. Still, development of applications using middleware is a complex task [32].

Depending on the middleware chosen for developing the software system, the system's functional and non-functional properties (e.g., synchronous or asynchronous communication, tight or loosely coupled interaction, security, scalability, etc...) may be more or less direct to enforce. For instance, a synchronous communication requirement would not be satisfied by using MOM as is. Similarly, using a CB middleware for developing a messaging application implies to explicitly deploy a message broker component and interact with it for sending and receiving messages, and to force asynchronous multicast communication by adding new components (e.g., proxies) that decouple message sender and receivers.

The diversity of properties impacted by middleware, both qualitative and quantitative, makes the development of high quality middleware-based software systems more complex than it appears and affects all the phases of the software development process. Consequently, software engineering methods and tools should be developed with the use of middleware in mind. In general, the use of middleware affects all development phases:

- Requirements engineering shall drive selection of the middleware.
- System architecting shall account for architectural constraints posed by the middleware, in terms of component deployment and interaction behavior.
- System design, specification and analysis must integrate properties managed at the middleware layer, possibly enabling confronting different middleware technologies.
- System implementation shall build as much as possible on middleware tool support, aiming at automating as far as possible generation of the code interfacing with the middleware.
- System validation needs to be performed according to the integration of middleware-related and application-specific components [7].

3. Middleware Challenges

While the development of legacy middleware has been significantly driven by requirements of distributed information systems, the ongoing evolution of the networking environment leads to a much broader application of distributed computing. As a result, new requirements arise for middleware, as illustrated by today's distributed applications like those surveyed below.

3.1. Today's Distributed Computing

A major source of the evolution of distributed computing relates to the now pervasive networking. Convergence of telecom and computer networks, widespread adoption of the Internet, and availability of broadband and wireless networks make network connectivity embedded in most digital resources. Two application areas that exploit such network connectivity, and further constitute a significant driver of middleware research, are grid computing and ubiquitous computing. The former illustrates evolution of Internet-based computing and the latter highlights the introduction of the computing infrastructure in environments other than the professional domains.

Grid computing

Grid computing originally arose for scientific purposes [23]. It addresses the creation of distributed communities that share resources such as storage space, sensors, software application and data, by means of a persistent, standard-based service infrastructure. In this setting, resource sharing concerns a direct and coordinated access to resources in order to achieve collaborative problem-solving. To this extent, Grid defines three fundamental requirements:

- *Coordination:* a Grid offers facilities for integrating and coordinating the distributed resources and their utilization by assuring non-functional properties such as security and policy.
- *Standard protocols:* a Grid offers services such as authentication and, resource discovery and access, by using standard protocols and interfaces rather than proprietary ones.

- *Quality of Service:* a Grid offers quality of service management that enables the delivery of services which meet the expected quality of service.

A Grid-oriented middleware then addresses the above requirements. It is developed by using standard technologies and provides proper abstractions and infrastructures for dealing with coordination and quality of service provisioning.

Ubiquitous computing

The vision of having "disappearing technologies", pioneered by ubiquitous computing [54], is now a reality thanks to the drastic evolution of software and hardware technologies over the last decade (e.g., wireless and sensor networks, mobile computing, agents). This vision encompasses the idea of building smart environments where computing and communication facilities are "everywhere", being embedded in most objects of the surrounding, and can be seamlessly accessed "every time". Then, "disappearing" means using such technologies while not being aware of them.

In order to realize the vision, distributed systems need to meet the following, still challenging requirements [45]:

- *User mobility:* in relation with "everywhere" and "every time" accessibility, the system should allow user mobility providing support for mobile networking and information access.

- *User intent:* with respect to making technologies "disappearing", the system shall have the ability of capturing, tracing and exploiting the users' intents in order to anticipate their need and proactively provide assistance.

- *Context awareness and adaptation:* still for the sake of disappearing technologies, the system shall be able to sense the context and possibly adapt its behavior accordingly so that users always experience the best quality possible (e.g., changing the user interface modality from speech to text display on the user's handheld in a noisy environment).

- *Delegation:* ubiquitous computing environment is typically composed of highly heterogeneous digital resources, with possibly limited computing and communication capabilities. Still to make computing capabilities always available, the system should support delegation of computing tasks from tiny-scale to wealthy resources.

- *High-level energy management:* a key enabler of ubiquitous computing is wireless networking. However, this makes energy a critical resource, which needs to be managed comprehensively, possibly leading to dynamically adapt computation.

- *Privacy and Trust:* satisfying the requirements listed above needs intensive management and spreading of user information. This requires both to trust the environment and to assure privacy in order to avoid undesired use of sensitive data.

A middleware for ubiquitous computing should be able to provide proper abstractions for the above requirements, while managing the highly heterogeneous and dynamic networking environment.

Towards next generation middleware

Grid and ubiquitous computing both illustrate major evolution of distributed software systems, i.e., systems are increasingly deployed over open networks that integrate computational resources from multiple administrative domains. However, software systems enabling grid and ubiquitous computing differ from their intended uses. On the one hand, grids define uniform global distributed infrastructures for use by applications (typically, scientific applications). On the other hand, ubiquitous computing is concerned with seamlessly delivering high quality services to end users. As a result, various distributed infrastructures have emerged for ubiquitous computing, leading to poor interworking. This thus suggests convergence of grid and ubiquitous computing as the ubiquitous grid [18], with the former defining uniform distributed infrastructures and the latter opening new application domains for grids, further posing additional requirements on the supporting middleware.

3.2. Open and Mobile Networking

With the advent of pervasive networking, thanks to the widespread deployment of both Internet and wireless network access, resources coordinated by distributed systems are no longer fixed at design time. As illustrated by the ubiquitous grid, systems shall now be deployed in open networks and access dynamically located resources. Openness is actually twofold in today's networking environments: (i) Internet connectivity allows networking with resources across network boundaries, and (ii) wireless networking allows resources held by mobile users to join and leave networks according to the users' mobility patterns. In both cases, distributed systems need be developed according to an abstract characterization of the networked resources to be coordinated, so as to allow late binding.

Open networking has been among early focus of middleware design, with the definition of the trader functionality

for dynamically discovering software (sub)systems according to given required properties. The trader typically defines a software repository available at a known location, together with protocols for publishing and retrieving systems from the repository [15]. The trader functionality further generalized into Service Discovery Protocols (SDP), which define an integrated architecture for resource discovery and access, and introduce a dedicated middleware. Using SDPs, software resources can be hosted and/or requested by mobile nodes, while participating nodes do not need to have any a priori knowledge of the networking environment that they may join and leave at will [10]. Yet another proven approach to overcome network openness is loosely coupled coordination, as offered by tuplespace and MOM-based middleware. In this way, messages sent over the network will reach and be consumed by relevant resources. Still, the high scale and dynamics of today's networks challenge the above legacy solutions to openness. Indeed resources need to be able to coordinate in heterogeneous environments, from the application to the network layers. Furthermore, coordination must be supported in a decentralized way to avoid reliance on a specific infrastructure, whose accessibility cannot always be guaranteed in mobile and open networks.

Another issue that arises in the effective coordination of networked resources relates to reasoning about their behavior. Relevant behavior further concerns both functional and non-functional properties. Abstract characterization of the behavior of networked resources has always been a key part of middleware. Middleware platforms define IDLs so that resources can be checked against requirements in terms of syntactic matching relationship over respective interface definition. However, this assumes a priori knowledge of the interfaces of networked resources, together with related behavior. This is a too strong assumption for open networking environments where resources freely join and leave networks from distinct administrative boundaries. In general, it cannot be assumed that resources that have matching functional behavior will have syntactically matching interfaces. This calls for adopting comprehensive software engineering approaches to software reuse based on the formal specification of software component behavior [55]. Still, common specification languages need to be adopted across networking environment, which can only be achieved through the emergence of standards. In this direction, effort in the area of Semantic Web services technologies is a promising direction. Openness of the network further raises trust concerns, i.e., the confidence that one may have about the actual behavior of networked resources. Service Level Agreements (SLAs) that define contracts associated with resource access provides sound base ground towards tackling this issue [50]. However, assisting the development of software resources, from their formal specification to SLA-aware deployment is still lacking adequate SE methods and tools.

Obviously, openness and dynamics of the network challenge enforcement of non-functional properties relevant to both dependability and performance concerns. In particular, such properties can no longer be handled in a systematic and transparent way. Overall, this has led to question the transparency of distribution advocated in the early days of distributed computing. The distributed application software needs to be informed about the networking environment in which it operates as it is in the best position to know how to change its behavior so that it still provides the intended service despite changes in the environment. Application-aware adaptation was in particular introduced to adapt to changes in the availability of computing system resources in mobile computing environments [44]. However, as stressed by the ubiquitous computing vision, distributed software systems need to adapt to various changes in the environment, covering not only changes in available system resources but also the profiles of its users over time and the physical environment impacting upon the user interfaces (e.g., lighting, sound). This ability of software systems is generically referred to as context-awareness.

3.3. Context-awareness

With network connectivity being pervasive and computing resources being embedded in most objects of our surroundings, context-awareness is becoming a key characteristic of distributed software systems. Indeed, sensing the physical environment and further adapting the behavior of applications according to both the users' profiles and available resources shall form a primary concern of software system development. Obviously, context-awareness is central to ubiquitous computing that aims at delivering applications to end-users in an opportunistic way, with the best quality possible. Such a concern is actually relevant to most software systems, as it promotes usability by a large diversity of users and enables system deployment and access in open and mobile environments.

Context-aware software systems specifically have the ability to seamlessly adapt their behavior according to the context within which the system executes. Resulting system adaptation may take several forms, spanning: changing internal processing, altering the content that is processed and exchanged over the network, and modifying user interfaces. Context information that may advantageously impact upon the behavior of software systems are numerous and can be classified into three major *context domains* [47]: (i) the *user domain* provides knowledge about the users via profiling, (ii) the *system domain* describes digital, software and hardware resources available to the execution of the software system, and (iii) the *environmental domain* deals with the description of location and of conditions of the physical en-

vironment.

Development of context-aware systems lies in: (i) *context management*, from sensing the environment for relevant context information to storing related context data in a machine-interpretable way, and (ii) *context-based adaptation* that specifies adaptation of the software system according to contextual changes. Middleware is thus key for assisting the development of context-aware software systems. Middleware can conveniently embed generic support for context management and making context data available to the application layer. Additionally, the middleware may itself be context-aware so that middleware functions adapt according to context like available network bandwidth [13].

Initial research on context-aware software systems focused on building applications for a specific scenario or a specific type of context, as exemplified by location-aware systems (e.g., see [53]). Due to their specific nature, these solutions provide limited support to generic context management that can be reused across context-aware software systems. Then, to ease the development of context-aware systems, dedicated frameworks have been proposed. Those frameworks provide reusable *context components* that are responsible of data acquisition, aggregation and interpretation (e.g., [20]). Still, making systems context-aware requires software development using the specific frameworks and further having the specific context components actually deployed in the environment. Availability of context information may then greatly be improved through the deployment of context servers within the network(s) together with an infrastructure for sensing and reporting context to the context server [29]. More recent solutions extend the server-oriented approach to context management to a decentralized architecture. In this way, every computing device (PDA as well as desktop) becomes responsible of managing part of context information and peering with others to enrich the context knowledge base [2]. Still, although the development of context-aware software systems has been an active area of research for more than a decade, significant advances are still needed so that it can be adopted for most applications [24].

First, context management shall be based on the opportune networking and cooperation of context sources, so as to make the context knowledge base available to applications as rich as possible. In particular, context sources shall now integrate latest sensor technologies, which raise specific requirement upon software engineering methods and tools [30]. The various context sources may then be coordinated by, possibly hierarchical, context repositories – if any – for the sake of resource savings. However, the actual cooperation protocol among context sources shall be determined dynamically according to various non-functional attributes, like availability, performance and privacy. Similarly, access to context sources (possibly via the repository) may use different protocols according to requirements for context knowledge (e.g., proactive, reactive, timely, decentralized). The context knowledge need further be exploited at all layers of the context-aware software system.

Context-aware systems shall adapt their behavior according to context. We specifically distinguish 3 types of context-sensitivity: (i) *context-specific systems* can only be correctly provisioned in the specified context, (ii) *context-dependent systems* may be provisioned in various contexts but are bound to a specific context during a session, i.e., adaptation is performed at access time, and (iii) *context-adaptive systems* continuously adapt their behavior according to context evolution. The third type of systems relates to the development of dynamically adaptive software [57], which is raising numerous challenges for the software engineering domain, e.g., how to program and automate as far as possible the adaptation dimension. Whatever is the type of context-sensitivity, specification of context-aware systems must include that of the context in which the system can be correctly executed, leading to context-aware analysis. Furthermore, the context must be monitored during the system execution by the middleware, for triggering adaptation – if supported – or enforcing robustness of the system by detecting that the system can no longer provisioned its intended behavior. This further calls for the system validation to be context-aware.

In general, the development of context-aware systems requires adaptation of software engineering methods and tools. Context must become a first class attribute in the definition of the systems' functional and non-functional properties. In addition, middleware shall play a pivotal role in enabling context-awareness by managing: (i) the context sensed from the multitude of networked resources and (ii) consistent adaptation of applications according to specification.

4. Next Generation Middleware-based Software Systems

The above requirements for middleware resulting from the significant and fast evolution of the networking domain are being tackled in the research community. This leads to a new generation of middleware-based software systems for which we identify key characteristics in this section.

4.1. Universal Interoperability

Middleware has been introduced to overcome heterogeneity, in the underlying hardware and software, of networked systems. By specifying a reference communication protocol, including message format and coordination model, middleware enables compliant software systems to

interoperate. Still, interoperability is achieved up to compliance to the specific middleware. Further, the emergence of different middleware, to address requirements of specific application domains and/or networking infrastructures, leads to a heterogeneity issue among communication protocols. The diversity of communication protocols is a key concern for open and mobile networked environments, where networked software resources can not all be known in advance. Indeed, it is not possible to predict the communication protocol to be used for accessing networked resources and, at a given time and/or at a specific place, devices hosting the wrong middleware become isolated.

Middleware-layer interoperability

A number of systems have been introduced to provide middleware protocols interoperability [38]. These include middleware bridges that provide interoperability between two given middleware by translating related communication protocols from one to another. Greater flexibility to bridge-based interoperability is brought by Enterprise Service Buses (ESBs) [12]. An ESB is a server infrastructure that acts as an intermediary among heterogeneous middleware through the integration of a set of reusable bridges. Yet another flexible approach to protocol interoperability relies on reflection, which allows dynamically plugging-in the most appropriate communication protocols according to the protocols sensed in the environment [28]. Event-based translation of protocols initially introduced for dealing with protocol evolution [17], is another alternative to middleware protocol interoperability [9]. In general, various proven solutions exist for middleware protocol interoperability, each with respective drawbacks and advantages [10]. However, these address only interoperability at the middleware layer.

Enabling software engineering techniques

Besides the middleware layer, interoperability between networked software resources concerns their interface and behavior at the application layer. As already discussed, the SOA paradigm enables loose associations between networked software systems. Therein, the service interface specifies the inputs and outputs that a service may exchange with its environment, in terms of data and structure, while the service behavior specifies valid sequences of such exchanges, as supported by the service.

The service interface and behavior can be described by employing appropriate specification languages. For the widely used Web Services SOA technology, a base interface description language (WSDL) has been introduced, as well as a number of workflow-like languages for describing service conversations, orchestrations and choreographies (e.g., BPEL[1], WS-CDL[2]). Such languages commonly have formal foundations that enable reasoning on the correctness of systems composed out of individual services , thus enabling interoperability between service-based systems. Specification formalisms and reasoning methods also draw from software engineering approaches to specification matching of software components [55].

However, in dynamic open networked environments, interoperability approaches cannot rely solely on syntactic descriptions of involved systems, for which common understanding is hardly achievable in such environments. Then, a promising approach towards addressing the interoperability issue relies on semantic modeling of information and functionality, that is, enriching them with machine-interpretable semantics. This concept originally emerged as the basis for the Semantic Web[3] [6]. Semantic modeling is based on the use of ontologies and ontology languages that support formal description and reasoning on ontologies; the Ontology Web Language (OWL)[4] is a recent recommendation by W3C. A natural evolution to this has been the combination of the Semantic Web and Web Services into Semantic Web Services [36]. This effort aims at the semantic specification of Web Services towards automating Web services discovery, invocation, composition and execution monitoring. The Semantic Web and Semantic Web Services paradigms address application-level interoperability in terms of information and functionality [52, 40].

From the above, it may be concluded that a comprehensive approach to system interoperability at application layer should support system description and reasoning upon this description at both semantic and syntactic level, for both the system interface and behavior. Further, related run-time interoperability mechanisms are required [5]. Software engineering for interoperability should then integrate adequate description languages and modeling methods – such as the ones discussed above – as well as related development tools for semantics- and behavior-aware service engineering.

4.2. Open Coordination

The scale of the networking environment within which distributed software systems execute, is drastically increasing and further allows coordinating with nodes that are a priori unknown in a possibly very short time period. As a result, traditional coordination models like client-server interactions are no longer adequate abstractions for the networking environment.

[1] http://www.ibm.com/developerworks/webservices/library/ws-bpel/
[2] http://www.w3.org/TR/ws-cdl-10/
[3] http://www.w3.org/2001/sw/
[4] http://www.w3.org/TR/owl-semantics/

Middleware-layer coordination

The above concern has been the topic of tremendous research over more than a decade, in the context of distributed computing over wide-area and wireless networks, which already posed requirements for open and mobile networking. Further, the scale and high dynamics of the networking environment, as envisioned by Beyond 3^{rd} Generation networks, only exacerbate the requirements of mobile computing [42]. As a result, adequate coordination abstractions need to be provided to application developers, thanks to underlying middleware support that manages corresponding overlay network. Most promising approaches are then those that tolerate highly dynamic networks, such as *content-based* and *peer-to-peer* networking.

Content-based networking [14] introduces a coordination model where messages flow from a producer to one or more consumers within the network, based on the preferences on the content of messages expressed by the consumers rather than by relying on explicit addresses. That is, a producer publishes messages, while consumers submit predicates for defining the messages they are interested in. The content-based network is then responsible for selecting those messages that satisfy the predicates, and for delivering them to the consumers. As depicted in Figure 3, a content-based network is built at the application-level and consists of an overlay network composed of client and router components. Clients can be either producer or consumer, while routers can act as both pure-routers and access-points. Both clients and routers can reside anywhere and be deployed autonomously within the environment by self-arranging into an overlay-network, without relying on intermediation by a (logically) centralized coordinator. These components do not have any specific network address and the interactions between them are connectionless and best-effort. These features make the content-based model a powerful and flexible communication style for building dynamic distributed systems, as required by next generation networking environment. In fact, loose coupling between the interacting parties allows them either to be easily added/removed (in the case of mobility), or to modify the binding by creating new messages and predicates as well as by modifying the already submitted ones.

Peer-to-Peer systems identify a class of software systems that addresses the direct access and sharing of computer resources without relying on intermediation by a (logically) centralized coordinator. According to [1], a peer-to-peer system is defined as a distributed system where the interconnected nodes are able to self-organize in an overlay-network (see Figure 4) for sharing resources such as CPU cycles, storage and bandwidth. Main characteristic of these systems is the ability of self-adapting to failures and accommodating transient nodes while providing high performance. The fact of not having central point of coordination, requires the nodes involved in the overlay-network to autonomously perform a number of tasks such as discovering other nodes (neighbors) and connecting to them, routing messages, locating and retrieving resources as well as providing security and performance means. The characteristic of being able to self-organize in spontaneous and heterogeneous overlay-network makes peer-to-peer networking a powerful communication infrastructure for dynamic distributed systems. In fact, the peer-to-peer networking independence of DNS and their total autonomy from administrative concerns, allow for operating in environments characterized by unstable connectivity and unpredictable IP addresses.

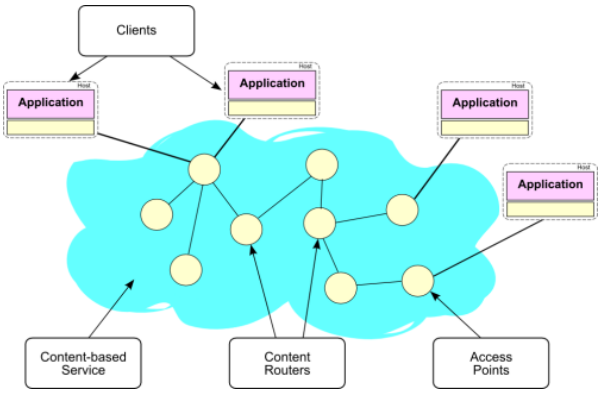

Figure 3. Content-Based Networking

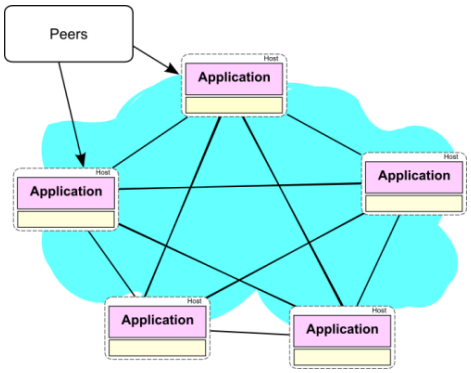

Figure 4. Peer-to-peer networking

Enabling software engineering techniques

Although the flexibility and openness of the above coordination models are two welcome characteristics for today's networking environments, they require for accurate system engineering.

In fact, content-based applications call for novel software engineering methods and tools for their systematic design, implementation and validation. Software engineering

methods are emerging in that direction, e.g., model-based verification [4, 12]. However, comprehensively dealing with the dynamics of the communication model remains an open issue. In particular, since the interactions between subsystems depend on both predicates and messages content, they can be established and verified only at runtime. Moreover, the system is affected by a number of non-functional properties (i.e., non-determinism in the messages ordering and possible delays in messages delivery) that are inherent to loosely coupled infrastructures.

As for content-based networking, the intrinsic dynamics of the peer-to-peer coordination model makes the design, implementation and validation of applications harder tasks. In fact, since the interactions between subsystems depend on the actual overlay-network (in terms of topology and node functionalities), they can be established and thus validated only at runtime. Furthermore, the presence of transient nodes makes the environment highly unstable, possibly hampering the system behavior.

4.3. Context-aware Adaptability

As already discussed, the distributed computing infrastructure now involves a high number of technologies with heterogeneous means and requirements. This then poses a new important requirement upon the application layer, namely, that it should be context-adaptive, i.e., have the ability to adapt to the current context in order to provide a possibly stable level of service despite changes in the operating conditions.

Context-adaptive applications can be categorized according to the type of adaptation they exploit [43]: (i) *customizable* exploit adaptation at compile time, (ii) *configurable* provide adaptation facilities at deployment time, (iii) *tunable* are able to be fine-tuned at runtime, and (iv) *mutable* may support radical changes at runtime. Obviously, tunable and mutable adaptations become desirable properties for dealing with the continuously evolving networking environments in which distribute systems operate, as promoted by the "anytime and anywhere" ubiquitous computing vision.

Middleware-layer adaptation

Due to the dynamic nature of the adaptation process, the role of middleware is central. First, middleware needs to adapt its behavior according to context (i.e., available computing and networking resources and application requirements). Second, middleware must provide relevant feedback about the underlying infrastructure to the application layer.

In general, middleware cannot rely on any a priori knowledge about the context to which it should adapt. Then, it should deal with heterogeneity of both the networking and computing environment, and the relative requirements. Further open issues due to the adaptation include [8]:

- Vertical coordinated system adaptation within a group of collaborating (mobile) nodes (e.g., peer-to-peer data sharing, shared workspaces and multimedia conference applications), where the peers should agree on a consistent view of the employed middleware mechanisms.
- Horizontal coordinated system adaptation within a single node across the various layers of the software architecture (from network protocols to application), e.g., through application-defined policies. In fact, the user and the application (as well as the user and application context) are often the most appropriate to make informed choices, which are based on high-level contextual or semantic information about how a system should adapt. Then, it is reasonable that the user and the application help drive the adaptation of the system.
- Dynamic system adaptation, further complicated by applying overlapping adaptations to several aspects (e.g., behavioral and structural), can result in system inconsistency. Therefore, ensuring safe system adaptation is a key issue.

Enabling software engineering techniques

Depending on where and when adaptation is implemented, it can be achieved by means of different software engineering techniques. At design time, *software design patterns* and *component-based design* are convenient paradigms. At runtime, *computational reflection* and *aspect-oriented programming* serve structuring the adaptation process.

Software design patterns are a general software engineering methodology for representing well-known solutions to common and recurring problems in software design [49]. In particular, design patterns are templates representing best software design practices, to be successfully applied for resolving new issues. Patterns have been exploited in providing flexible and reusable solutions for developing adaptive systems.

Component-based design [51] is a well-known programming paradigm that can be exploited for developing flexible and extensible software systems. In fact, a system can be customized to specific application domains, through the integration of domain-specific components. Further, component-based software systems can be dynamically adapted to their environment by using late composition.

Computational reflection [35] is a programming technique that allows an application to inspect and reason about its own internal status and possibly alter its behavior accordingly. In particular, two different types of reflection may be

distinguished: (i) *structural reflection*, which deals with the underlying structure of objects or components (in terms of supported interfaces); and (ii) *behavioral reflection*, which deals with the underlying system activity (in terms of the arrival and dispatching of invocations). As already indicated in Section 4.1, reflection has already been proven successful in adapting distributed systems in terms of protocols used for remote interactions.

Aspect-Oriented programming (AOP) [33] is a software engineering approach that enables separation of cross-cutting concerns, such as QoS, fault tolerance, security, logging and monitoring, during development time. Such concerns are not implemented within the base code implementing functional features; rather, they are each implemented as an individual *aspect*, which is a piece of code that can then be woven into the base code at either compile or run time. There are two techniques for dynamically inserting aspects into software systems: *invasive* and *non-invasive*. Invasive dynamic AOP breaks the component architecture by weaving code within the base component implementation, i.e., behind the interface contracts, whereas non-invasive approaches use the component interfaces as point-cuts, and hence these aspects are implemented as interceptors on the interfaces; the latter is closely related to behavioral reflection.

Both AOP and computational reflection approaches address the development of generic applications that can be adapted at runtime according to the sensed context (i.e., available resources and their own characteristics and status). Enforcing the correctness of such adaptive applications is a crucial software engineering task. That is, once having developed the generic adaptable application code, its correct adaptation with respect to a given execution context must be inferred [31].

On the other hand, component-based design and software design patterns address system adaptation in a compositional way. In that direction, Software Architecture (SA) is a well-known and effective tool for describing and modeling complex systems in terms of component composition [26]. SA enables describing both the static and dynamic composition of the system subparts, as well as their interactions; it can therefore be exploited for achieving system adaptation [39].

4.4. Middleware-based Software Process

As discussed in Section 3, the vision of the future computing infrastructure is towards a global virtually loosely-connected world with invisible computers everywhere embedded in the environment. Exploiting both mobility and availability of a potentially infinite number of heterogeneous resources at the same time recalls well known requirements such as scalability and resource-discovery, and poses new ones such as context-awareness and adaptability. Furthermore, considering the continuous evolution of technology (in terms of both software and hardware) and the proliferation of new functionalities, the middleware cannot be considered anymore the place where diversities are homogenized in a transparent way (as discussed in Section 2) but where diversities are discovered and exploited in a fully aware modality. This raises a number of software engineering challenges such as the need of defining new languages and methods for describing and modeling systems throughout the software development process, as well as tools that allow for validating the developed systems.

For all of the above reasons, we consider middleware having an increasing central role in the overall software development process. Hence, its own properties and functionalities should be exploited since the early stages of the software life-cycle. This drives towards the need of defining a middleware-based software development process where both functional and non-functional characteristics of the middleware are taken into account during all phases of the software life-cycle. That is, the entire system should be thought, designed, analyzed, implemented, deployed and validated by having the middleware in mind.

Model-Driven Engineering (MDE) is a significant step towards a middleware-based software process [48]. MDE refers to the systematic use of models as primary artifact for the engineering of systems [25]. The best known MDE initiative is the Model-Driven Architecture (MDA) from the Object Management Group (OMG), which is intended to software systems. Using the MDA methodology, system functionalities may first be defined as a platform-independent model (PIM) through an appropriate Domain Specific Language. Given a Platform Definition Model (PDM) corresponding to some middleware, the PIM may then be translated to one or more platform-specific models (PSMs) for the actual implementation. The translations between the PIM and PSMs are normally performed using automated tools, like Model transformation tools [34]. In general, MDE appears as a useful approach towards reasoning about the impact of middleware on the behavior of software systems. It further enables automating part of the development process. Still, supporting methods and tools are in their infancy.

Figure 5 depicts the phases (represented by squared boxes) of the overall middleware-based software development process and their productions (represented by ovals). Based on the results coming from the *Requirements* elicitation phase, the first step consists in selecting a middleware platform that properly addresses the application domain concerns. Both the application domain and the selected middleware affect the selection/creation of the *conceptual model* upon which building the application. In fact, since the conceptual model role is to define application con-

cepts and their relationship to each other, it has to take into account the concepts strictly related to the application as well as the middleware ones. This allows engineering to actively consider the middleware, along with its high-level characteristics (e.g., context awareness and adaptability), during the development process, and to benefit of those low-level aspects usually abstracted away (e.g., network functionalities and coordination model).

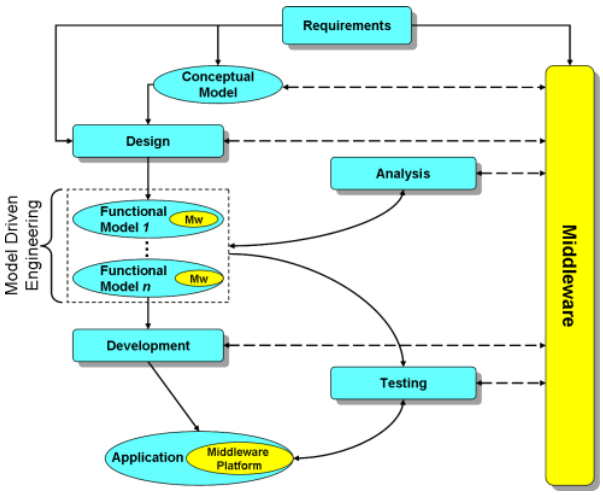

Figure 5. Middleware-based Software Process.

During the *Design* phase, the application will be then designed considering both the conceptual model and the constraints (in terms of structure and interaction) posed by the middleware. Performing model-to-model transformations, defined by MDE, a number of models are generated to refine the initial one as well as to build specific models useful for analysis purposes.

Exploiting model-based analysis techniques, the *Analysis* phase is in charge of validating the application design against both functional (e.g., behavioral [12]) and non-functional (e.g., performance [3]) properties. Also in this case, the middleware characteristics have to be considered since they can affect the application validation. Model-based analysis techniques can also be used for driving the *Test* phase [37].

Once the design has been validated, during the *Implementation* phase, the actual application is built on top of the middleware platform by exploiting all its characteristics. The resulting runtime code is finally tested within the *Test* phase.

The key idea, on which we envisioned such a development process, is that in order to fully exploit all the features provided by next-generation middleware platforms, they should be considered "core" elements of development process rather than simply a "mean/tool". In fact, as extensively discussed in previous sections, middleware plays a central and essential role for implementing next-generation applications and its characteristics (both functional and non-functional) must be considered in a fully aware modality in order to get benefit from them.

5. Conclusion

Since its emergence, middleware has proved successful to assist distributed software development, making development faster and easier, and significantly promoting software reuse. Basically, middleware contributes to relieve software developers from low-level implementation details, by: (i) at least abstracting socket – level network programming in terms of high-level network abstractions matching the application computational model, and (ii) possibly managing networked resources to offer quality of service guarantees and/or domain specific functionalities, through reusable middleware-level services. The advent of middleware standards further contributed to the systematic adoption of the technology for distributed software development. However, as noticed in [22] and [46], mature engineering methodologies to comprehensively assist the development of middleware-based software systems, from requirements analysis to deployment and maintenance, are lagging behind. Indeed, software development accounting for middleware support is only emerging [48]. Still, the networking infrastructure is evolving at a fast pace, and suggest new development paradigms for distributed systems, leading to new middleware platforms and further calling for novel software engineering methods and tools. Major system requirements posed by today's networking infrastructure relate to openness and mobility, and context-awareness. This leads to investigate next generation middleware with support for universal interoperability, open coordination, and context-aware adaptability. Further, with the middleware becoming central in software development, software processes shall evolve so as to integrate middleware features at all phases of the software development.

Acknowledgments

This work is part of the PLASTIC[5] and AMIGO[6] IST projects, funded by the European Commission under the FP6 contract numbers 026955 and 004182 respectively.

References

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing. Survey*, 36(4):335–371, Dec. 2004.

[5] http://www.ist-plastic.org/
[6] http://www.hitech-projects.com/euprojects/amigo/

[2] AWARENESS Consortium. http://awareness.freeband.nl.

[3] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[4] L. Baresi, C. Ghezzi, and L. Mottola. Towards fine-grained automated verification of publish-subscribe architectures. In *Proceedings of FORTE*, 2006.

[5] S. Ben Mokhtar, N. Georgantas, and V. Issarny. COCOA: Conversation-based service composition in pervasive computing environments. In *Proceedings of IEEE International Conference on Pervasive Services*, Lyon, France, Jun 2006.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[7] A. Bertolino. Software testing research: Achievements, challenges, dreams. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[8] G. Blair. Open middleware architecture and adaptation. MiNEMA summer school, Klagenfurt University, Austria., Jul 2005.

[9] Y.-D. Bromberg and V. Issarny. INDISS: Interoperable discovery system for networked services. In *Proceedings of the 6th International Middleware Conference*, Grenoble, France, Nov. 2005.

[10] Y.-D. Bromberg, V. Issarny, and P.-G. Raverdy. Interoperability of service discovery protocols: Transparent versus explicit approaches. In *Proceedings of the 15th IST Mobile & Wireless Communications Summit*, Mykonos, June, Jun. 2006.

[11] M. Caporuscio. *Design, Development and Analysis of Distributed Event-Based Systems*. PhD thesis, Dept. of Computer Science, University of L'Aquila, Jan. 2007.

[12] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, May 2004.

[13] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Transactions of Software Engineering*, 2003.

[14] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *Proceedings of NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, Arizona, Oct 2001.

[15] R. Y. M. Cheung. From grapevine to trader: the evolution of distributed directory technology. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, Ontario, Canada, 1992.

[16] H. Cunningham and G. Roman. A unity-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), Jul 1990.

[17] N. D. Ryan and A. L. Wolf. Using event-based translation to support dynamic protocol evolution. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, May 2004.

[18] N. Davies, A. Friday, and O. Storz. Exploring the grid's potential for ubiquitous computing. *IEEE Pervasive Computing*, 3(2), Apr-Jun 2004.

[19] N. Desai, A. K. Mallya, A. Chopra, and M. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12), Dec. 2005.

[20] A. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[21] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.

[22] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, Jun. 2000.

[23] I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[24] D. Fournier, S. Ben Mokhtar, N. Georgantas, and V. Issarny. Towards ad hoc contextual services for pervasive computing. In *Proceedings of the International Workshop on Middleware for Service Oriented Computing*, Melbourne, Australia, Nov. 2006.

[25] R. France and B. Rumpe. Model-driven development of complex systems: A research roadmap. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[26] D. Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, Jun. 2000.

[27] D. Gelernter. Generative communication in linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.

[28] P. Grace and G. B. S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2005.

[29] 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), 2005.

[30] S. Hadim and N. Mohamed. Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3), 2006.

[31] P. Inverardi, F. Mancinelli, and M. Nesi. A declarative framework for adaptable applications in heterogeneous environments. In *Proceedings of the 19th ACM Symposium on Applied Computing*, 2004.

[32] V. Issarny, C. Kloukinas, and A. Zarras. Systematic aid for developing middleware architectures. *Communications of the ACM - Issue on Adaptive Middleware*, 45(6), Jun. 2002.

[33] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Jun 1997.

[34] A. Kleppe. Mcc: A model transformation environment. In *Proceedings of the 2nd European Conference on Model Driven Architecture Foundations and Applications*, Bilbao, Spain, Jul. 2006.

[35] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages*, Orlando, Florida, United States, Dec. 1987.

[36] S. Mc Ilraith and D. Martin. Bringing semantics to web services. *IEEE Intelligent Systems*, 18(1), 2003.

[37] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE Transactions on Software Engineering*, 30(3):160–171, 2004.

[38] J. Nakazawa, H. Tokuda, and W. K. Edwards. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, 2006.

[39] P. Oreizy, M. M. Golick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[40] D. O'Sullivan and D. Lewis. Semantically driven service interoperability for pervasive computing. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, San Diego, California, USA, Sep. 2003.

[41] M. Papazoglou and D. Georgakopoulos. Special section: Service-oriented computing. *Commun. ACM*, 46(10), 2003.

[42] G. Roman, G. Picco, and A. L. Murphy. Software engineering for mobility: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, Jun. 2000.

[43] S. Sadjadi and P. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, Dec 2003.

[44] M. Satyanarayanan. Accessing information on demand at any location. *Mobile Information Access. IEEE Personal Communications*, Feb. 1996.

[45] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[46] R. E. Schantz and D. C. Schmidt. Research advances in middleware for distributed systems: state of the art". In *Proceedings of IFIP World Computer Congress*, 2003.

[47] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, 1994.

[48] D. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), Feb 2006.

[49] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*. John Wiley, 2001.

[50] J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proceedings of the 26th Int.ernational Conference on Software Engineering*, Edinburgh, UK, May 2004.

[51] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[52] A. Tsounis, C. Anagnostopoulos, and S. Hadjiefthymiades. The role of semantic web and ontologies in pervasive computing environments. In *Proceedings of Mobile and Ubiquitous Information Access Workshop*, Glasgow, UK, Sep. 2004.

[53] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10, 1992.

[54] M. Weiser. The computer for the 21^{st} century. *Scientific American*, Sep. 1991.

[55] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), Oct 1997.

[56] A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware. In *Proceedings of the International Middleware Conference*, 1998.

[57] J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the International Conference on Software Engineering*, May, Shanghai, China 2006.