1. Introduction

The technological development of robotics research will soon lead to the marketing of robots that can play a key role in supporting people in their everyday tasks. Pursuing a specific objective while dealing with a dynamic environment and ensuring a safe interaction with human beings, requires a complex multifunctional structure for robot control, where heterogeneous hardware and software components interact in a coordinated manner. Additionally, further requirements are being introduced by an increasing number of projects [1,2] adding cognitive requirements while preserving pervasive requisites of autonomous robotics design, i.e., the capability to have a real-time interaction with the real world [3].

The robotics community has recently proposed several architectures for the development of robot control software [4–9]. This includes the avoidance of monolithic development methodologies since they are unable to deal with the problem complexity. Despite the large number of significant proposals, there is still a lack of common, suitable solutions that would allow the reuse of previous efforts. The main reason for this failure is the difficulty of clearly describing and formally defining a problem domain which is still unclear in the field of multifunctional robots: for the same problem, different research projects still produce different specifica-tions for its domain. This also holds for cognitive robotics research where projects only share a common understanding of cognition as the ability to think or reason about embodiment worlds, but there are quite different assumptions about the representation, organization, utilization, and acquisition of knowledge. This has a huge impact on the final software architectures as it often prevents the exchange of software solutions developed by different research groups.

Even if the robotics community is still not in the stage of avoiding the recreation of incompatible solutions, a plague which is common to other software research fields, it would greatly benefit from the advances and maturity reached by distributed technology research. This research field is already converging toward a few technical architecture paradigms, and mature implementations of these ideas are freely available in the form of software middlewares supporting complex interprocess communication, event synchronization, and data distribution. A thoughtful application of these research results in the development of robotic software architectures would, at least, alleviate the cost of re-invention of core concepts and techniques for the control of distributed devices. Nevertheless, their application to robotics research is still late, often relying only on the basic concepts of the available middlewares.

In this paper, we shortly introduce three technical architecture paradigms that have been successfully exploited in the robotics field (Section 2). Their characteristics and successful stories within the robotic domain are discussed in detail in Sections 3–5. We discuss the benefits and tradeoffs of the different solutions with the goal of deriving some practical principles and strategies to be

Architectural paradigms for robotics applications

Michele Amoretti, Monica Reggiani

A R T I C L E   I N F O

Article history:
Received 12 June 2009
Accepted 5 August 2009
Available online 2 September 2009
Handled by Prof. I. Smith

A B S T R A C T

In recent years, several technical architectural paradigms have been proposed to support the development of distributed and concurrent systems. Object-oriented, component-based, service-oriented approaches are among the most recent paradigms for the implementation of heterogeneous software products that require complex interprocess communications and event synchronization. Despite the sharing of common objectives with distributed systems research, the robotics community is still late in applying these research results in the development of its architectures, often relying only on the most basic concepts.

In this paper, we shortly illustrate these paradigms, their characteristics, and the successful stories about their application within the robotic domain. We discuss benefits and tradeoffs of the different solutions with the goal of deriving some practical principles and strategies to be exploited in robotics practice. Understanding the characteristics, features, advantages, and drawbacks of the different paradigms is, indeed, crucial for the successful design, implementation, and use of robotic architectures.

© 2009 Elsevier Ltd. All rights reserved.

1474-0346/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2009.08.004
exploited in robotics practice. Understanding the characteristics, features, advantages and drawbacks of the different paradigms is indeed crucial for the successful design, implementation, and use in robotic architectures. Finally, we present a set of guidelines with an in-depth discussion about the influences and impacts of architecture paradigms on robotic applications to drive the design of control software architecture for robotics projects (Section 6).

This paper focuses on the analysis of technical architecture paradigms and software strategies for their use in the robotics domain. A performance comparison of the final control software architectures is outside of the paper’s objectives. Interested readers can refer to other papers dealing with the comparison robotics architectures [10,11] and to the middleware activity of the Rosta project [12].

2. Architectural paradigms

The development of complex cognitive embodied systems is a challenging task as it requires a collection of behavior control abilities, including perception, manipulation, and learning. These abilities have to work concurrently and have to collaborate through the exchange of available knowledge. The design of intermodule communication and event synchronization is therefore of main importance during the development of the control software infrastructure. Several design methodologies and architectural paradigms for data communication have been proposed by the distributed computing community. This section is a short review that will introduce the evolution process that brought the development of the three technical architectural paradigms discussed in this paper.

Fig. 1 illustrates several abstraction layers, of increasing complexity, for distributed applications. At least one paradigm lays on each layer, i.e. a pattern or model defining the best design practices.

At the lowest level, Message Passing is the fundamental paradigm for distributed applications and provides an abstraction to encapsulate the details of the network communication and the operating system. Intermodule communications are based on send and receive primitives that allow input/output in a manner similar to the file I/O.

One abstraction layer consists of Message-Oriented Middleware (MOM) and Remote Procedure Call (RPC), which are two of the most prominent communication paradigms [13]. In the MOM paradigm, a message system serves as an intermediary among separate, independent modules. The message system acts as a switch for messages, allowing modules to exchange messages asynchronously, in a decoupled manner. Using a Point-to-Point communication model, MOM forwards a message from the sender to the receiver’s message queue. Compared to the basic message-passing model, this paradigm provides the additional abstraction for asynchronous operations. Their support with message passing would have required low-level implementation through threads or child processes. Another MOM communication model is Publish/Subscribe which, at each message, associates a specific topic, task, or event. Modules interested in the occurrence of a specific event may subscribe to messages for that event. When the event occurs, the process publishes a message announcing the event or topic and the MOM message system distributes the message to all the subscribers.

The second paradigm of this layer, the Remote Procedure Call (RPC), allows distributed software to be programmed like conventional applications which run on a single process. A Remote Procedure Call causes a subroutine or procedure to be executed in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. The programmer, therefore, writes the same code whether the subroutine is local or remote in respect to the executing process.

An increasing request for modularity and abstraction is what drove the development of the three architectural paradigms in the last abstraction layer. The Distributed Object Architecture (DOA) paradigm (Section 3) is based on the object oriented approach and is an improvement over the first attempts to provide platform independent solutions for interprocess communication. In particular, remote method invocation is the object-oriented equivalent of Remote Procedure Calls, where the remote object takes the role of the remote process. In this model, a process invokes the methods in a (remote) object, which may reside in a remote host. As with RPC, arguments may be passed along with the invocation. A following step introduced the concept of software components [14] created with the objective of promoting the reuse of design and implementation efforts. The final objective of the Component Based Architecture (CBA) paradigm (Section 4) is the development of components, eventually from multiple sources, that can be deployed according to customers’ needs, often evolving during the project’s lifetime. A recent trend for the development of modern large-scale distributed and mobile systems is calling for a new solution that will be better able to support an automated use of the available distributed resources. The idea of presenting software as a service is at the base of the Service-Oriented Architecture (SOA) paradigm (Section 5). The SOA has been recently introduced to provide loosely coupled, highly dynamic applications.

The previously described paradigms address several needs in abstraction granularity for the development of distributed applications. Another significant problem is to guarantee the reliability and efficiency of the whole distributed system by choosing the most scalable overlay scheme. The client/server scheme assigns asymmetric roles to the collaborating processes. One process, the server, plays the role of resource provider, passively waiting for request arrivals. The other (client) issues specific requests to the server and awaits its replies. The peer-to-peer (P2P) paradigm, envisions direct resource sharing among participants having close capabilities and responsibilities. Whereas the client/server paradigm is an ideal model for centralized robotic applications, such as teleoperations, the peer-to-peer paradigm is more appropriate for cooperative robotics, swarm robotics, and ambient intelligence.

In the next section, we focus on high-level solutions for the communication problem, introducing the basic characteristics of the DOA, CBA and SOA paradigms together with some of the most representative examples of their application in the robotics domain. This will lay the background that is required to motivate the choice to apply the different paradigms when a new robotic application must be developed.
3. Distributed object architecture

Distributed Object Architecture (DOA) concepts are the result of the merging of object-oriented design techniques with distributed computing systems. According to the definition provided by the Object Management Group (OMG) (http://www.omg.org), DOA applications are “composed of objects, individual units of running software that combine functionality and data”, and run on multiple computers to act as a scalable computational resource. To support the interaction between server-side objects and clients invoking them, DOA systems rely on the definition of interfaces. Each distributed object must declare its interface, i.e. the available operations used by clients to identify the requests supported by the object, and by the DOA system to implement the marshaling/unmarshaling of the operation arguments. As an evolution of object-oriented techniques, DOA developers often identify fine-grained interfaces which result in multiple object interactions requiring high levels of control on concurrency.

3.1. DOA standards and middlewares

Among the several DOA proposals in the latest 15 years, the Common Object Request Broker Architecture (CORBA) has achieved the highest level of maturity and diffusion. CORBA (http://www.corba.org) is a vendor-independent specification promoted by the OMG. CORBA overcomes the interoperability problem allowing the smooth integration of systems that have been built using different software technologies, programming languages, operating systems, and hardware. To support portability, reusability, and interoperability, CORBA defines the Object Request Broker (ORB), a fundamental component that behaves as a system bus, and connects objects operating in an arbitrary configuration (Fig. 2). To achieve language independence, CORBA requires developers to define how clients will make a request using a standard and neutral language: the OMG Interface Definition Language (IDL). After the interface is defined, an IDL compiler automatically generates client stubs and server skeletons according to the chosen language and operating system. Client stub implementation produces a thin layer of software that isolates the client from the Object Request Broker, allowing distributed applications to be transparently developed from object locations. The Object Request Broker is in charge of translating client requests into language-independent requests using the Generic Inter-ORB Protocol (GIOP). It is also in charge of communicating with the Server through the Internet Inter-ORB Protocol (IIOP) and of further translating the request in the language chosen at the server side. Together with the Object Request Broker, the architecture proposed by OMG introduces several CORBA services, providing capabilities that are needed by other objects in a distributed system.

3.2. DOA robotic applications

CORBA is in wide use as a well-proved architecture for building and deploying significant robotics systems. The use of CORBA is growing in this sector because it can harness the increasing number of operating systems, networks, and protocols while supporting real-time scheduling into an integrated solution that provides end-to-end QoS guarantees into distributed object applications. Initial robotics projects using CORBA took a simple approach to ORB technology. They often ignore fundamental components such as the Naming Service for location transparency [15], or they exploit CORBA only for the interoperability of previously developed components [16]. Following these experiences, other investigations used CORBA to achieve interoperability and location transparency in their applications and to exploit other useful CORBA Services [17–21].

Several recent projects developing robot architectures have based their work on CORBA. Miro [5] is an object-oriented robot framework that is freely available as an open source. It supports multiple robotics platforms and common operating systems and provides a set of interfaces for communication among objects. The overall infrastructure is largely based on a client/server view built upon certain standards and on widely used CORBA packages to simplify the integration of different robotics tasks. Humanoid control architectures have also used CORBA for the implementation of communication layers. The large number of often heterogeneous hardware and software components that compose a humanoid are already a distributed architecture that can benefit from distributed middlewares to simplify software development [22–24].

The main area of application of DOA technology is currently in the development of real-time and embedded systems. Stringent requirements about computing resources and time constraints have pushed for improvement on the efficiency, scalability, and predictability of DOA middlewares implementations [25]. The availability of Real-Time CORBA ORBs allows to develop systems that use multithreading while controlling the amount of memory and processor resources they consume [26–28].

4. Component-based architecture

Component-based architectures (CBA) are built upon the concept of a software component, i.e. a unit of composition with a contractually specified interface [29]. Following the DOA approach, CBA forces a strong separation between the interface and implementation to simplify the design of large systems and promote software reuse. Nevertheless, DOA objects are not good candidates for CBA components. While objects need to be tightly coupled with each other to achieve their functionality, components should be autonomous units whose purpose is well defined and understood. As a consequence, components are generally more coarsely-grained than objects.

Usually CBA approaches define a model that the component developers have to follow in order to allow graceful composition. This model specifies the creation, use, and lifecycle management of components and includes a programming model for their definition, assembly, and deployment. Interactions can follow several schemes (synchronous, asynchronous, event-driven, etc.), and they are usually not statically defined but can be manipulated at run-
have been built upon CBA principles. Indeed, the component-based approach proposes a possible solution for several weaknesses on the robotics software. One problem refers to the great effort usually required for the development and setup of control software for robotic platforms that will then be used for the implementation and evaluation of research issues. The aim of the CBA approach is to develop components for mature algorithms, sensors, and actuators that can be easily downloaded or purchased and flexibly combined. Another problem relates to distributed environments, providing location transparency for easy component rearrangement on processing and bandwidth constraints, which are often required by the latest robotics applications. Additional details about CBA and robotics can be found in Ref. [32].

Among the several available proposals, two mature projects are RT-Middleware [33,34], based on OMG concepts, and ORCA [35]. RT-Middleware is a developmental framework created at the National Institute of Advanced Industrial Science and Technology (AIST). Its main goal is to simplify system integration through a methodology for the creation of Robotics Technology Components (RT-Components) and a framework for their composition. RT-Components, built as CORBA components, consist of the following objects and interfaces: Component Object, Activity, continuously processing inputs; InPort as an input port object, OutPort as an output port object, and Command Interface. RT-Middleware supports several methods to integrate RT-Components, such as an assembly GUI tool, a script language, and XML configuration files. The AIST research laboratory has also developed OpenRTM-aist, a prototype implementation based on RT-Middleware interface specification and the RT-Component model, used to develop several testbeds such as a force controlled manipulator system [34], a service robotic system for elderly care [36], and an image recognition device [37]. ORCA is an open-source implementation framework for developing component-based robotic systems [6]. The main objective of ORCA is to provide the tools for defining and developing the components that will be combined together to support the implementation of an arbitrary robotic architecture. ORCA achieves this goal through the adoption of a component-based approach. The definitions of the interfaces and communications are based on the Internet Communication Engine (ICE) middleware [38]. Meanwhile additional tools have been developed to support the implementation of the components retaining full access to the underlying details. Through the identification of common definitions for data structures and interfaces that are frequently encountered in robotics, ORCA can build a repository of reusable components, libraries, and utilities [39].

5. Service-oriented architecture

Service-oriented computing defines a paradigm whose goal is to achieve loose coupling among interacting software entities, thus minimizing artificial dependencies. The key concept of this paradigm is the service: a unit of work executed by a service provider to achieve the results desired by a service consumer. Provider and consumer are both simply roles played by software entities on behalf of their owners. Therefore, service consumers are considered as end users, provided with client tools, or other services. The interaction pattern among service providers and consumers is illustrated in Fig. 4. The most important achievement of the SOA-based distributed environments is that shared resources (mainly applications and data) are available on demand as independent services that can be accessed without knowledge of their underlying platform implementation.

5.1. SOA standards and middlewares

A good starting point for understanding the SOA paradigm is OWL-S [40,41], a service ontology that supplies a core set of mark-
up language constructs for describing services in unambiguous, computer-interpretable form. This would allow the automatic discovery, invocation, composition, cooperation and execution monitoring of services. OWL-S is attracting a lot of interest even though it is still under development and suffers some conceptual ambiguity and lack of concise axiomatization.

Meanwhile, and even before the rise of OWL-S was driven by the Semantic Web Community, SOA applications have mainly been created and deployed using Web Services. This technology aims at moving beyond the traditional middleware and framework concepts, standardizing higher-level interaction patterns, service flow orchestration and enterprise application integration. A number of protocols and standards define Web Services. Web Services Description Language (WSDL) documents describe the Web Service interface through the identification of the supported operations and messages and through their binding to a concrete network protocol and message format. Web Service interfaces are usually listed in centralized repositories, such as UDDI registries, but there is still no standard protocol for distributed publication and discovery of Web Services.

The loose coupling between consumers and providers is achieved through a stateless request/reply scheme for a message-oriented interaction. Messages are typically conveyed using Simple Object Access Protocol (SOAP), i.e. HTTP with an XML serialization, but any other communication protocol could be used for message delivery. For example, REST (Representational State Transfer) is being used in several new Web Service applications. REST basically dictates that each unique URL is a representation of some resource for information, which can be managed using simple HTTP messages. In respect to SOAP Web Services, REST Web Services are lightweight, with a reduced XML markup, are easier to build, and the results are human readable. Finally, the Web Services Resource Framework (WSRF) specification has been recently introduced to support the creation of stateful Web Services.

The Web Services platform-neutral technology has been implemented on several platforms that support their development and deployment. J2EE and .NET are the most successful and will be briefly introduced in the following sections.

### 5.1.1. Web Services with J2EE

Among the several J2EE competing environments, the most widely used is JBoss [42] which provides the whole range of J2EE features. JBoss includes extended enterprise services including clustering, caching, and persistence, as well as a J2EE certified platform for the development and deployment of enterprise Java applications, Web applications, and portals. The open source community also provides important toolkits for the Java-based development of Web Service architectures. Apache products are the most notable, ranging from Web Service containers to specific protocols implementations. Tomcat [43] is the servlet container belonging to the Apache suite and it is used in the official reference implementation for the Java Servlet and JavaServer Pages technologies. Axis [44] is instead the SOAP engine, i.e. a framework for the construction of SOAP processors such as clients, servers, gateways, etc. Axis version 2 supports both SOAP and REST.

### 5.1.2. Web Services with .NET

Windows Communication Foundation (WCF) is the Microsoft platform for SOA [45]. It is a rich technology foundation that aims at building distributed service-oriented applications for the enterprise and the web. The latest version of .NET (3.0), officially launched with Windows Vista in January 2007, introduced WCF along with Windows Workflow Foundation to support service composition. This marked the release of the first Microsoft Web Services platform for the design, implementation and deployment of services with essential plumbing for scalability, performance, security, reliable message delivery, transactions, multithreading, and asynchronous messaging.

WSRF.NET is another important set of libraries, tools, and applications which implement the WSRF specifications. This free software was developed by the Grid Computing Group of the University of Virginia and allows easy authoring of WSRF-compliant services and clients and integrates many Microsoft technologies [46].

Recently, Microsoft released the Microsoft Robotics Studio (MRS) [47] a software based on .NET that provides a service-oriented architecture combining key aspects of traditional Web-based architectures with new concepts from Web Service technologies. The MRS runtime adopts the REST model as its foundation, and extends it with structured data representation and event notifications from the Web Service world. MRS supports several programming languages, including those in Microsoft Visual Studio (C# and VB.NET) and scripting languages such as Python.

### 5.2. SOA robotic applications

In the first phase, the adoption of SOAs into distributed robotic applications was limited to the wrapping of existing applications with services, with limited exploitation of SOA protocols and tools [48–51]. Recently, the research community entered a second phase in which applications are being (re)designed according to service-centric models, considering also advanced specifications such as OWL-S and WSRF.

In this context, Ha et al. [52] proposes the automated integration of distributed robots, sensors, and devices into ubiquitous computing environments based on semantically enriched Web Services. Their Ubiquitous Robotic Service Framework (USRF) consists of three major components: a Robotic Agent (RA), an Environmental Knowledge Repository (EKR), and Device Web Services (DWS). The RA includes a service application, a URSF Application Programming Interface (API), a plan composition module, a knowledge discovery module, a plan execution module, an OWL reasoner, and a protocol stack for Web Service execution including SOAP, XML and HTTP. To request a service from a robot, a user inputs a command for the service application through a user interface. The service command is encoded in OWL-S profile ontology and the concept ontology is stored in the EKR to enable the understanding...
of the users’ command through the knowledge discovery and plan composition modules.

The service-oriented architecture for Web Labs proposed by Coelho et al. [53] is targeted to education applications. In this architecture, the building blocks are services that can be recursively composed to produce more comprehensive services. Lab resources (physical and logical) are modeled and implemented as services, e.g., a robot exports a set of services, each one performing a specific function (sensing, navigation, etc.). The concept of the federation of services allows Web Labs to use resources maintained by other Web Labs located in different administrative domains.

Saffiotti et al. [54] explore a reactive approach to self-configure an ecology of robots. This was inspired by ideas from the field of semantic Web Services, but the resulting middleware (called PEIS-kernel) is neither based on J2EE nor .NET technologies. Their work is a clear example of how SOA principles can be decisive for solving complex distributed robotic problems. The proposed approach presents three main characteristics. First, there is a formal description of functionalities so they can be exported to the ecology and be automatically processed. Second, starting from the description of the current task, a framework is available for runtime finding and the composition of functionalities exported by different robots and is required to solve the objective task. Finally, a mechanism for semantic interoperability allows the matching of functionalities from heterogeneous devices according to a unified logical classification.

The last application we consider is the healthcare robot platform introduced by Lee et al. [55], based on a Web Service Event–Condition–Action (WS-ECA) framework. ECA rules consist of events, notification messages from services or users, conditions, Boolean expressions that must be satisfied to activate devices and actions which are instructions that involve services or generate events. The healthcare robot platform is equipped with various sensors, including ultrasonic sensors for distance measurement, infrared human detection sensors, and navigation sensors. It can collect vital signals from bio-sensors, such as heart rate, blood pressure, and breath rate. These sensors are active publishers of context events, which can be registered by the WS-ECA engine as operators. This ECA-based approach is becoming widely used in ambient intelligence applications.

### 6. Evaluation criteria for architectural paradigms selection

The development of any robotics software application should start with a clear definition of its purpose and scope, the use cases to be fulfilled, and the required quality attributes [56]. The purpose and scope determine the intended users, boundaries with other systems, operating environments, and properties of application domains. Use cases are the first step towards the definition of the functional architecture, providing high level description of user-robot, robot-robot, and robot-environment interactions. Finally, quality attributes specify the required levels of performance, flexibility, extensibility, sustainability, openness, and interoperability that the system should provide.

This initial effort for application specifications is critical for the follow-up steps because it guides developers to make sound architectural design decisions. This phase is even more critical for cognitive robotics applications since quite different functional specifications can be recognized within the research field on cognitive systems, requiring different levels of abstractions, i.e. different levels of granularity and performance.

We will identify the key concepts that should undergo rigorous analysis to guide the developers who are interested in the application of the presented technical architectural. The main concepts and principles of DOA, CBA, and SOA could look similar, indeed, but each one has its unique approach, characteristics, features, and benefits. This section goes into detail about the differences among DOA, CBA and SOA, developing an in-depth discussion about the influences and impacts of architectural paradigms on robotics applications, following similar investigations in other research fields [56–58]. With respect to these works, we developed an analysis that is driven by robotics rather than by general software engineering needs. Table 1 shows the introduced key concepts and summarizes the following discussion.

#### 6.1. Specification and granularity

Specification and granularity are tied aspects for the presented architectural paradigms. They both refer to module interfaces defining either the level of supported granularity or how to create the description of the module abstraction. As most systems require the interaction among several distinct modules [59], an improper definition of these concepts can seriously affect the development process and the final performance. When the application is composed of several robotic systems, they often are required to interact to transfer knowledge among them [60]. This usually requires a clear definition of what each system can provide (services/knowledge/information) and how it can be requested. Even when an isolated robot is considered, its architecture is usually composed of several distinct modules that require the integration of deliberative cognitive models with software and hardware engineering methodologies and techniques [59]. These modules are usually finely grained as they control low-level processes of sense stimuli from the environment to extract information for the high-level deliberative behaviors. The three paradigms offer a rather different approach to granularity and specification. Objects allow the system to define the robotic application with fine granularity and they are therefore suitable for the implementation of control architecture on both deliberative (planning and scheduling) and reactive layers. Instead, component interfaces (component contracts) provide a high level list of operations and context dependencies, allowing larger granularity. In between, service interfaces (service descriptions) allow medium granularity through information such as the service signature, expected behavior, and quality attributes.

#### 6.2. Coupling

Coupling is the degree to which each program module relies on the other modules [61]. Tight coupling causes a system to be hard to modify, because each change will usually result in other required changes, in a domino effect. This introduces complexity since the process of discovering appropriate changes is both time consuming and error-prone: a network of interdependencies makes it hard to see, at a glance, how the modules work. With loose coupling, instead, a change in one module will not require changes in the implementation of other modules.

DOAs are usually affected by tight coupling since clients must be adapted to remote objects on each interface change. When possible, coupling among objects can be partially relieved by the adoption of an event-based scheme for data distribution [62]. The CBA paradigm introduces some level of coupling. As a component is used within the scope of a component model, it needs to conform to its specified rules. Usually, a component model uses a particular interaction style, such as broadcasting, asynchronous connection, or connection-oriented style. These interaction styles imply some coupling among components such as referential or temporal coupling. Finally, a SOA paradigm supports loose coupling among interacting services through two architectural constraints. First, a small set of simple and ubiquitous interfaces, with only generic semantics encoded, represents a contract between the clients and the services. Second, descriptive messages are constrained by an
extensible scheme delivered through the interfaces. Any, even only minimal, system behavior is prescribed by the messages. The schema only limits the vocabulary and structure of the messages and allows new service versions to be introduced without affecting existing services.

6.3. State

A paradigm for architectures supporting cognitive systems requires the availability of stateful modules. Indeed, to operate in an environment, an intelligent system should be able to make decisions and selections among alternatives. This is supported by prediction mechanisms that require knowing the state of the system to make decisions. Storing the description of the current situation therefore requires stateful modules that are able to represent this information in the memory [57].

The DOA paradigm is tailored for systems whose modules are stateful entities. As the DOA module type (object) is an instance of a class, it always includes a state defined by the current values of its attributes. Classes without attributes are feasible, but they would be a contradiction because a class should be a blueprint of a factory that describes the nature of something (a robot, sensor, image, sound, etc.). Components and services can either be stateless or stateful. In a stateless form, each time a calling module requires a task execution, the stateless component is instantiated and lasts until task accomplishment. In a stateful form, a component remains instantiated after the execution of a task and until the application specifically terminates it; thus, information is retained between separate component calls. There are both advantages and disadvantages associated with stateless and stateful software components. Stateless components have less system resource overhead due to their short existence and an absence of state information. However, communication traffic may increase toward stateful modules because task calls also require the dispatching of additional information for task instantiation. There are cases in which it is useless to have a stateful service. Consider a service that returns the current value of a property of the environment measured by a sensor; this is a simple reactive behavior that does not require any memory. The use of stateless services can improve performance and, sometimes, the stateless property is optimal for reusability.

6.4. Workflow

Many cognitive robotics applications ask for efficient communications to support knowledge transfer. Prediction mechanisms require information from the environment through perception, knowledge from other agents via direct communication, and awareness of past experience through memory and learning [63,64]. Therefore, an in-depth analysis of the required workflow, i.e. the way a task is executed by the interacting modules, is preliminary to the choice of the architectural paradigm.

DOAs support either an explicit message passing between task controllers, or an event channel at which distributed objects register [65]. The first approach leads to tightly coupled systems. The task controllers or the task managers need to be aware of other dependent task controllers and task managers. Any change in the workflow logic, then, has to be propagated to all the relevant workflow components in the system. With the event-channel approach, the workflow is effectively enacted by the reaction and generation of new events for which inter-task dependencies do not have to be explicitly registered in the distributed workflow components. Tasks in the workflow react differently to different events, e.g. based on an event–condition–action (ECA) rule logic could be common to different tasks, hence offering greater re-usability of workflow logic. In CBAs, workflow is usually managed by a set of software components including a process definition component, an enactment component (Engine), and a work list handler [52]. Each work list describes the implementation of a task by composing specific activities executed by different components. In SOAs, there are two main approaches servicing workflow execution, namely orchestration and choreography. An orchestration model provides a scope specifically focused on the view of one participant. Orchestration allows the design of a central entity (the orchestrator) which carries out a business activity invoking other services. For instance, if there are two services which are required to be synchronized, the first has to send the synchronization message to the orchestrator engine which will forward it to the latter. The orchestrator also stores the states of the activities it is carrying out. A choreography model encompasses all parties and their associated interactions, giving a global view of the system. Choreography aims at constraining the behaviors of the services involved in the system by regulating the exchange of their messages. Moreover, the state of each activity is distributed among the entities.

6.5. Reusability

Another important criteria to evaluate cognitive robotics requirements are reusability. An architecture should be versatile, requiring a reduced effort for its adaptation to new environments and tasks. As shown in Refs. [66,59], several programming environments have been proposed in robotics, with different approaches for robotic systems development and integration. This introduced several difficulties related to their reuse when moving from one paradigm to another.

In DOAs, the reusability of objects is supported by two basic OOP mechanisms: inheritance and polymorphism. Moreover, a large number of design patterns for distributed and networked objects have been defined (see the five POA books [67]). Many DOA programming environments, such as Player [68], Carmen [69], Miro [5], and CLARaty [7], have solved this problem by choosing specific communication protocols and/or mechanisms that need to be implemented by all applications to be linked together. MARIE [59], based on the CBA paradigm, proposes another solution adapt-
ing the Mediator design pattern [70] for distributed systems. The Mediator is distributed between all the components (robots, parts of robots, devices) thus realizing a virtual space. Each component must be adapted only to the Mediator, rather than to all the other components, thus allowing high reusability. In SOAs, the reusability of services is a factor that generates additional costs, as services must be developed in a way that they can be used not just for the current project but for other applications, too. In SOA applications, developers should construct the services to be as simple as possible, and refactor them so that they are as broadly applicable as possible. The resulting services are then reusable at runtime, nuggets of software functionality (both fine and coarse-grained) can be used in a variety of situations, as contrasted to trying to solve the issue of reuse at design-time. This principle also affects the granularity dimension: services should be “right grained”. In a SOA, fine-grained, atomic services should be composed into coarse-grained services, but in practice, different services have different levels of granularity, depending on their functionalities.

6.6. Extensibility

Extensibility is the systemic measurement of the level of effort required to extend a system to cope with a new range of problems and amounts of knowledge. Since robotic architectures must be used in practice, they must be quickly adaptable to new tasks in unexpected environment. Extensibility can be achieved through the addition of new modules to the system, through the inclusion of new functionalities to an existing module, or through the modification of existing functionalities.

In DOAs, distributed objects can be extended through inheritance, or by adding new objects. In the same way, SOAs can be extended by adding new services. Extending a service is reasonable if the service is a composition of atomic services whose extension requires the adding of new services to the workflow. The CBA technology is based on the notion that components are independently developed and deployed by unrelated parties. While component composition is common, component extensibility is often limited because mainstream class-based object-oriented programming languages, which are currently used for the development of software components, do not meet a number of important requirements. Examples [71] include high-level abstractions for components and composition mechanisms, modular encapsulation (as a high-level information hiding mechanism), parametric polymorphism (to support genericity on the component level), and subtype polymorphism (to enable substitutability and variability of software components).

6.7. Overhead

Autonomous robotics aims at supporting intelligent embodiments that operate within an environment that they must sense, perceive, and interpret to cope with unpredictable changes. To ensure system reactivity, the designer should carefully define the time and space constraints that must be satisfied, and minimize the related overheads. The implementation of the three presented architectural paradigms frequently introduces overhead in communication among modules, either when they are parts of the same robot or of networked robots composing a collaborating team.

DOAs and CBAs implementations are usually affected by low overhead while SOA overhead largely depends on the adopted technology. Given the overhead of encoding and decoding XML, it is not surprising that Web Services are an order of magnitude slower than that of distributed object implementations in CORBA or RMI. But SOA is more than Web Services and there are several SOA implementations using technologies that do not depend on remote procedure calls or translation through XML. Additionally, there are emerging, open-source XML parsing technologies, such as VTD-XML [72], and XML-compatible binary formats that promise significant improvements in SOA performance. Another option to reduce overhead is to use REST (Section 5), instead of SOAP, for messaging.

6.8. Example scenario

A goal of the next generation robots is the capability to adapt to users' needs. This is an expected requirement for both professional and domestic service robots, required to interact with people in a natural style of communication to support their activities. Recently, adaptation is becoming increasingly important in manufacturing robotics. Mass manufacturing is, indeed, less important for the new economy. Intelligence and ease of use are instead required in robots that must adapt to produce small runs of personalized products, customized to clients' requests. This scenario envisions an increasing number of highly pervasive robotic applications with several networked robots moving around in a smart environment which is able to grasp users’ situation and communicate this information to visible robots.

As an example of the previously mentioned applications, consider the following complex scenario: an unmanned vehicle system (UVS) exploring an unknown smart environment. Both the robot and the environment sense and act upon each others. The implementation of this scenario can greatly benefit from the advances of distributed technology research. Nevertheless, none of the technical architectural paradigms can, alone, sustain all information communication requirements. Due to the heterogeneity of the involved systems and requirements, the final software infrastructure should be a well-balanced mix of DOA-, CBA-, and SOA-based solutions.

An effectual design strategy could define the following two separated areas: the UVS control and the interaction among devices within the smart environment. The first one must correctly interfaces with the low-level sensors to acquire the required data for the updating of the system and the control of the robot. The second one, instead, should be able to manage the large amount of data from the devices and their distribution to the interested agents, and it should continuously provide a high-level distributed representation of the state of the environment.

The UVS system requires software supporting communication and data links, vehicle actuation, sensor control, and data management. This introduces stringent requirements on the interaction among the fine-grained modules. Therefore, the chosen paradigm must ensure a reduced overhead, such as the DOA and CBA. The tight coupling introduced by these paradigms is acceptable due to the low dynamic in the configurations of modules and interactions.

An example of a successful robotics architecture exploiting these paradigms is the Open Robot Control Software (OROCOS) [4]. Applications are built using the control component, a distributable entity with a control oriented interface. A component could be capable of controlling a whole machine, but usually it is just a single part connected to a network of other components, for example implementing an interpolator or kinematic component. OROCOS uses a CORBA implementation as the middleware for the distributed components. Nevertheless, OROCOS users can implement new components without any CORBA knowledge because its use is wrapped within OROCOS software.

The smart environment requires information exchange among a combination of robots, ubiquitous sensors, PCs, and mobile devices. The large amount of environmental data has to be distributed as a coarse grain stream of meaningful information in a highly dynamic change of environment components. Therefore,
the main attention moves from reducing overhead to improving the dynamic integration and management of such a complex system. Usually, the best solution is a lightweight CBA/SOA platform, as it provides loose coupling among interacting services, advanced workflow models, and the possibility to reuse the same software functionalities for different applications.

A state-of-the-art example is the PERSONA platform [73]. The components of a PERSONA system are interfaced through the PERSONA middleware providing different communication buses. In the abstract model, each bus is a standalone module, but in the concrete implementation it is distributed among components, adopting open peer-to-peer communication strategies. Once components are linked with the middleware, they register with communication buses to find each other and collaborate, i.e. sending notification of status changes or executing actions and providing status queries. PERSONA components can, therefore, be service providers and consumers as well as I/O and context publishers and subscribers.

The UVS and the smart environment have their own internal communication system. Reciprocal knowledge is achieved through explicit message exchange, either at the component or service level, and through sensors, requiring additional elaboration to extract useful information from a stream of raw data (Fig. 5).

7. Conclusions

We reviewed three main architecture paradigms, namely DOA, CBA, and SOA, and their influence and impacts on architectures for robotics applications. While their main concepts and principles could look similar each one has, instead, its unique approach, characteristics, features, and benefits. DOA is the result of the merging of object-oriented design techniques and distributed computing systems. It is fundamental for the design of object-oriented systems and it is often used to identify fine-grained interfaces that need a high level of control on concurrency during multiple objects interactions. CBA, on the other hand, is fundamental for the design of components that expose interfaces that are more coarsely-grained than objects that deploy autonomous units with well defined and understood purposes. Finally, SOA allows for the definition of loose coupling interacting software entities, independent services that can be accessed without knowledge of their underlying platform implementation.

Since these paradigms have different levels of abstraction, it is unlikely that they may be used indifferently during the implementation of new robotics applications. We proposed an in-depth discussion to clarify the concepts, principles, and characteristics of DOA, CBA, and SOA, as the first step for their efficient use in future robotics software architectures.

References
