Addressing Challenges of Distributed Systems using Active Components

Lars Braubach and Alexander Pokahr

Abstract The importance of distributed applications is constantly rising due to technological trends such as the widespread usage of smart phones and the increasing internetworking of all kinds of devices. In addition to classical application scenarios with a rather static structure these trends push forward dynamic settings, in which service providers may continuously vanish and newly appear. In this paper categories of distributed applications are identified and analyzed with respect to their most important development challenges. In order to tackle these problems already on a conceptual level the active component paradigm is proposed, bringing together ideas from agents, services and components using a common conceptual perspective. It is highlighted how active components help addressing the initially posed challenges by presenting an example of an implemented application.

1 Introduction

Technological trends like the widespread usage of smart phones and the increased internetworking of all kinds of devices lead to new application areas for distributed systems and pose new challenges for their design and implementation. These challenges encompass the typical *software engineering* challenges for standard applications and new aspects, summarized in this paper as *distribution*, *concurrency,* and *non-functional properties* (cf. [9]).

Distribution itself requires an underlying communication mechanism based on asynchronous message passing and in this way introduces a new potential error source due to network partitions or breakdowns of nodes. Concurrent computations are an inherent property of distributed systems because each node can potentially act in parallel to all other nodes. In addition, also on one computer true hardware concurrency is more and more available by the advent of multi-core processors. This concurrency is needed in order

Distributed Systems and Information Systems Group, University of Hamburg
{braubach, pokahr}@informatik.uni-hamburg.de

to exploit the available computational resources and build efficient solutions. Non-functional aspects are important for the efficient execution of distributed applications and include aspects like scalability and robustness.

In order to tackle these challenges different *software or programming paradigms* have been proposed for distributed systems. A paradigm represents a specific worldview for software development and thus defines conceptual entities and their interaction means. It supports developers by constraining their design choices to the intended worldview. In this paper *object*, *component*, *service* and *agent orientation* are discussed as they represent successful paradigms for the construction of real world distributed applications. Nonetheless, it is argued that none of these paradigms is able to adequately describe all kinds of distributed systems and inherent conceptual limitations exist. Building on experiences from these established paradigms in this paper the active components approach is presented, which aims to create a unified conceptual model from agent, service and component concepts and helps modeling a greater set of distributed system categories.

The next section presents classes of distributed applications and challenges for developing systems of these classes. Thereafter, the new active components approach is introduced in Section 3. An example application is presented in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2 Challenges of Distributed Applications

To investigate general advantages and limitations of existing development paradigms for distributed systems, several different classes of distributed applications and their main challenges are discussed in the following. In Fig. 1 theses application classes as well as their relationship to the already introduced criteria of software engineering, concurrency, distribution and non-functional aspects are shown. The classes are not meant to be exhaustive, but help illustrating the diversity of scenarios and their characteristics.

Software Engineering: In the past, one primary focus of software development was laid on *single computer systems* in order to deliver typical desktop applications such as office or entertainment programs. Challenges of these applications mainly concern the functional dimension, i.e. how the overall application requirements can be decomposed into software entities in a way that good software engineering principles such as modular design, extensibility, maintainability etc. are preserved.

Concurrency: In case of resource hungry applications with a need for extraordinary computational power, concurrency is a promising solution path that is also pushed forward by hardware advances like multi-core processors and graphic cards with parallel processing capabilities. Corresponding *multi-core* and *parallel computing application* classes include games and video manipulation tools. Challenges of concurrency mainly concern preservation of state consistency, dead- and livelock avoidance as well as prevention of race condition dependent behavior.

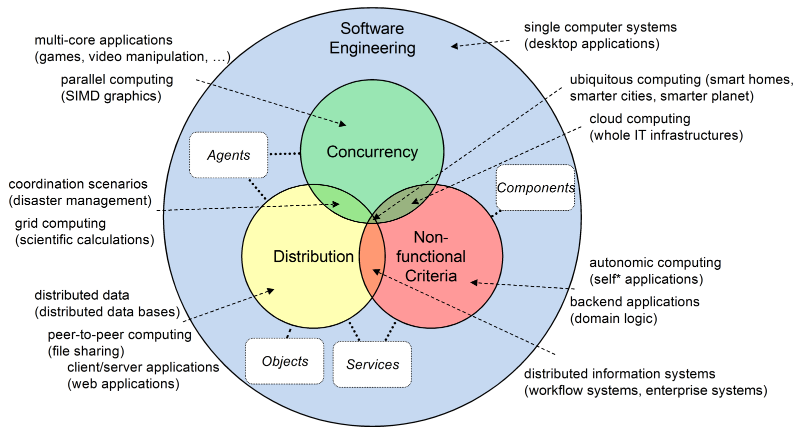
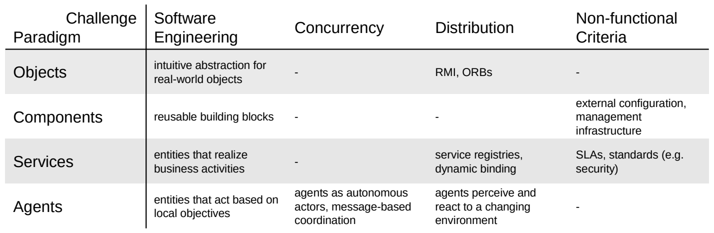

Software Engineering
single computer systems (desktop applications)
multi-core applications (games, video manipulation, ...)
parallel computing (SIMD graphics)
ubiquitous computing (smart homes, smarter cities, smarter planet)
cloud computing (whole IT infrastructures)
Agents
Concurrency
Components
coordination scenarios (disaster management)
grid computing (scientific calculations)
Distribution
Non-functional Criteria
autonomic computing (self* applications)
backend applications (domain logic)
distributed data (distributed data bases)
peer-to-peer computing (file sharing)
client/server applications (web applications)
Objects
Services
distributed information systems (workflow systems, enterprise systems)

Fig. 1 Applications and paradigms for distributed systems

Distribution: Different classes of naturally distributed applications exist depending on whether data, users or computation are distributed. Example application classes include *client/server* as well as *peer-to-peer computing* applications. Challenges of distribution are manifold. One central theme always is distribution transparency in order to hide complexities of the underlying dispersed system structure. Other topics are openness for future extensions as well as interoperability that is often hindered by heterogeneous infrastructure components. In addition, today's application scenarios are getting more and more dynamic with a flexible set of interacting components.

Non-functional Criteria: Application classes requiring especially non-functional characteristics are e.g. centralized *backend applications* as well as *autonomic computing* systems. The first category typically has to guarantee secure, robust and scalable business operation, while the latter is concerned with providing self-* properties like self-configuration and self-healing. Non-functional characteristics are particularly demanding challenges, because they are often cross-cutting concerns affecting various components of a system. Hence, they cannot be built into one central place but abilities are needed to configure a system according to non-functional criteria.

Combined Challenges: Today more and more new application classes arise that exhibit increased complexity by concerning more than one fundamental challenge. *Coordination scenarios* like disaster management or *grid computing* applications like scientific calculations are examples for categories related to concurrency and distribution. *Cloud computing* subsumes a category of applications similar to grid computing but fostering a more centralized approach for the user. Additionally, in cloud computing non-functional aspects like service level agreements and accountability play an important role. *Distributed information systems* are an example class containing e.g. workflow management software, concerned with distribution and non-functional aspects. Finally, categories like *ubiquitous computing* are extraordinary difficult to realize due to substantial connections to all three challenges.



Challenge / Paradigm	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Objects	intuitive abstraction for real-world objects	-	RMI, ORBs	-
Components	reusable building blocks	-	-	external configuration, management infrastructure
Services	entities that realize business activities	-	service registries, dynamic binding	SLAs, standards (e.g. security)
Agents	entities that act based on local objectives	agents as autonomous actors, message-based coordination	agents perceive and react to a changing environment	-

Fig. 2 Contributions of paradigms

Fig. 2 highlights which challenges a paradigm conceptually supports. Object orientation has been conceived for typical desktop applications to mimic real world scenarios using objects (and interfaces) as primary concept and has been supplemented with remote method invocation (RMI) to transfer the programming model to distributed systems. Component orientation extends object oriented ideas by introducing self-contained business entities with clear-cut definitions of what they offer and provide for increased modularity and reusability. Furthermore, component models often allow non-functional aspects being configured from the outside of a component. The service oriented architecture (SOA) attempts an integration of the business and technical perspectives. Here, workflows represent business processes and invoke services for realizing activity behavior. In concert with SOA many web service standards have emerged contributing to the interoperability of such systems. In contrast, agent orientation is a paradigm that proposes agents as main conceptual abstractions for autonomously operating entities with full control about state and execution. Using agents especially intelligent behavior control and coordination involving multiple actors can be tackled.

Yet, none of the introduced paradigms is capable of supporting concurrency, distribution and non-functional aspects at once, leading to difficulties when applications should be realized that stem from intersection categories (cf. Fig. 1). In order to alleviate these problems already on a conceptual level the active component paradigm is proposed in the following.

3 Active Components Paradigm

The active component paradigm brings together agents, services and components in order to build a worldview that is able to naturally map all existing distributed system classes to a unified conceptual representation [8]. Recently, with the service component architecture (SCA) [6] a new software engineering approach has been proposed by several major industry vendors including IBM, Oracle and TIBCO. SCA combines in a natural way the service oriented architecture (SOA) with component orientation by introducing SCA components communicating via services. Active components build on SCA and extend it in the direction of sofware agents. The general idea is to transform passive SCA components into autonomously acting service providers and consumers in order to better reflect real world scenarios which are composed

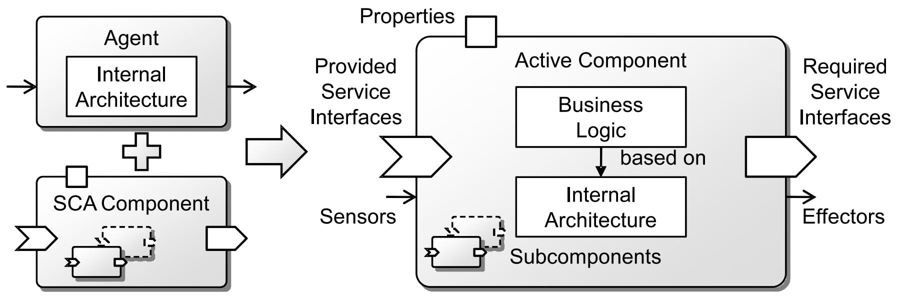


Fig. 3 Active component structure

of various active stakeholders. In Fig. 3 an overview of the synthesis of SCA and agents to active components is shown. In the following subsections the implications of this synthesis regarding structure, behavior and composition are explained.

3.1 Active Component Structure

In Fig. 3 (right hand side) the structure of an active component is depicted. It yields from conceptually merging an agent with an SCA component (shown at the left hand side). An agent is considered here as an autonomous entity that is perceiving its environment using sensors and can influence it by its effectors. The behavior of the agent depends on its internal reasoning capabilities ranging from rather simple reflex to intelligent goal-directed decision procedures. The underlying reasoning mechanism of an agent is described as an agent architecture and determines also the way an agent is programmed. On the other side an SCA component is a passive entity that has clearly defined dependencies with its environment. Similar to other component models these dependencies are described using required and provided services, i.e. services that a component needs to consume from other components for its functioning and services that it provides to others. Furthermore, the SCA component model is hierarchical meaning that a component can be composed of an arbitrary number of subcomponents. Connections between subcomponents and a parent component are established by service relationships, i.e. connection their required and provided service ports. Configuration of SCA components is done using so called properties, which allow values being provided at startup of components for predefined component attributes. The synthesis of both conceptual approaches is done by keeping all of the aforementioned key characteristics of agents and SCA components. On the one hand, from an agent-oriented point of view the new SCA properties lead to enhanced software engineering capabilities as hierarchical agent composition and service based interactions become possible. On the other hand, from an SCA perspective internal agent architectures enhance the way how component functionality can be described and allow reactive as well as proactive behavior.

3.2 Behavior

The behavior specification of an active component consists of two parts: service and component functionalities. Services consist of a service interface and a service implementation. The service implementation contains the business logic for realizing the semantics of the service interface specification. In addition, a component may expose further reactive and proactive behavior in terms of its internal behavior definition, e.g. it might want to react to specific messages or pursue some individual goals.

Due to these two kinds of behavior and their possible semantic interferences the service call semantics have to be clearly defined. In contrast to normal SCA components or SOA services, which are purely service providers, agents have an increased degree of autonomy and may want to postpone or completely refuse executing a service call at a specific moment in time, e.g. if other calls of higher priority have arrived or all resources are needed to execute the internal behavior. Thus, active components have to establish a balance between the commonly used service provider model of SCA and SOA and the enhanced agent action model. This is achieved by assuming that in default cases service invocations work as expected and the active component will serve them in the same way as a normal component. If advanced reasoning about service calls is necessary these calls can be intercepted before execution and the active component can trigger some internal architecture dependent deliberation mechanism. For example a belief desire intention (BDI) agent could trigger a specific goal to decide about the service execution.

To allow this kind service call reasoning service processing follows a completely asynchronous invocation scheme based on futures. The service client accesses a method of the provided service interface and synchronously gets back a future representing a placeholder for the asynchronous result. In addition, a service action is created for the call at the receivers side and executed on the service's component as soon as the interpreter selects that action. The result of this computation is subsequently placed in the future and the client is notified that the result is available via a callback.

In the business logic of an agent, i.e. in a service implementation or in its internal behavior, often required services need to be invoked. The execution model assures that operations on required services are appropriately routed to available service providers (i.e. other active components) according to a corresponding binding. The mechanisms for specifying and managing such bindings are part of the active component composition as described next.

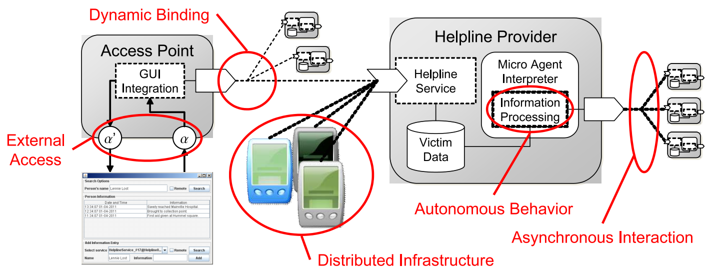


Fig. 4 Helpline system architecture

3.3 Composition

One advantage of components compared to agents is the software engineering perspective of components with clear-cut interfaces and explicit usage dependencies. In purely message-based agent systems, the supported interactions are usually not visible to the outside and thus have to be documented separately. The active components model supports the declaration of provided and required services and advocates using this well-defined interaction model as it directly offers a descriptive representation of the intended software architecture. Only for complex interactions, such as flexible negotiation protocols, which do not map well to service-based interactions, a more complicated and error-prone message-based interaction needs to be employed.

The composition model of active components thus augments the existing coupling techniques in agent systems (e.g. using a yellow page service or a broker) and can make use of the explicit service definitions. For each required service of a component, the developer needs to answer the question, how to obtain a matching provided service of a possibly different component. This question can be answered at design or deployment time using a hard-wiring of components in corresponding component or deployment descriptors. Yet, many real world scenarios represent open systems, where service providers enter and leave the system dynamically at runtime [4]. Therefore, the active components approach supports besides a static wiring (called *instance* binding) also a *creation* and a *search* binding (cf. [8]). The *search* binding facilities simplified specification and dynamic composition as the system will search at runtime for components that provide a service matching the required service. The creation binding is useful as a fallback to increase system robustness, e.g. when some important service becomes unavailable.

4 Example Application

The usefulness of active components is shown by describing an example system from the disaster management area, implemented using the Jadex framework.[1] The general idea of the *helpline* system, currently implemented as a small prototype, is supporting relatives with information about missing family members affected by a disaster. For information requests the helpline system contacts available helpline providers (hosted by organizations like hospitals or fire departments) and integrate their results for the user. The information is collected by rescue forces using mobile devices directly at the disaster site. The system can be classified as ubiquitous computing application and exhibits challenges from all areas identified in Section 2.

The main functionality of the system is implemented in the decentralized *helpline provider* components (cf. Fig. 4). Each component offers the *helpline service* allowing to access information as well as adding new information about victims into a database. To improve data quality, helpline providers perform autonomous *information processing*, by querying other helpline providers and integrating information about victims. This behavior is realized using the micro agent internal architecture, which includes a scheduler for triggering agent behavior based on agent state and external events. Simplified versions of the helpline provider components are installed in PDAs used by rescue forces, which synchronize newly added data with backend helpline providers, when a connection is available. To process user requests, *access points* issue information queries to all available helpline providers and present the collected information to the users.

The red markers in Fig. 4 highlight advantages of active components. They provide a natural metaphor for conceptually simplifying system development by supporting *autonomous behavior* and *asynchronous interaction*, which effectively hide details of concurrency and communication and thus reduce the risk of errors related to race conditions or deadlocks. The composition model allows for *dynamic binding* making it especially well suited for open systems in dynamic environments. On a technical level, the *distributed infrastructure* for active components provides a unified runtime environment with awareness features to discover newly available nodes that can also span mobile devices like android phones. Moreover, active components offer *external access* interfaces for easy integration with 3rd-party code, e.g. for integration in a web application or desktop user interface.

5 Related Work

In the literature many approaches can be found that intend combining features from the agent with the component, object or service paradigm. Fig. 5 classifies integration proposals according to the paradigms involved.

In the area of agents and objects especially concurrency and distribution has been subject of research. One example is the active object pattern, which represents an object that conceptually runs on its own thread and provides an asynchronous execution of method invocations by using future return values [11]. It can thus be understood as a higher-level concept for concurrency

[1] `http://jadex.sourceforge.net`

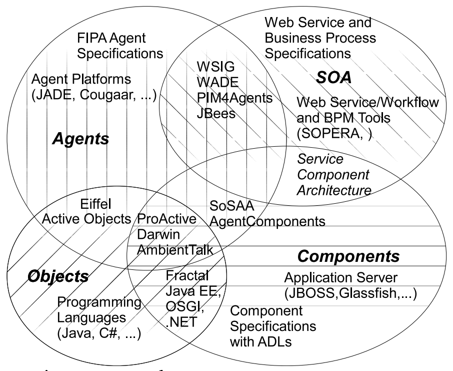

FIPA Agent Specifications
Agent Platforms (JADE, Cougaar, ...)
Agents
WSIG
WADE
PIM4Agents
JBees
Web Service and Business Process Specifications
SOA
Web Service/Workflow and BPM Tools (SOPERA,)
Service Component Architecture
Eiffel
Active Objects
ProActive
Darwin
AmbientTalk
SoSAA
AgentComponents
Components
Objects
Fractal
Java EE,
OSGI,
.NET
Application Server (JBOSS,Glassfish,...)
Programming Languages (Java, C#, ...)
Component Specifications with ADLs

Fig. 5 Paradigm integration approaches

in OO systems. In addition, also language level extensions for concurrency and distribution have been proposed. One influential proposal much ahead of its time was Eiffel [7], in which as a new concept the virtual processor is introduced for capturing execution control.

Also in the area of agents and components some combination proposals can be found. SoSAA [2] and AgentComponents [5] try to extend agents with component ideas. The SoSAA architecture consists of a base layer with some standard component system and a superordinated agent layer that has control over the base layer, e.g. for performing reconfigurations. In AgentComponents, agents are slightly componentified by wiring them together using slots with predefined communication partners. In addition, also typical component frameworks like Fractal have been extended in the direction of agents e.g. in the ProActive [1] project by incorporating active object ideas.

One active area, is the combination of agents with SOA [10]. On the one hand, conceptual and technical integration approaches of services or workflows with agents have been put forward. Examples are agent-based service invocations from agents using WSIG (cf. JADE[2]) and workflow approaches like WADE (cf. JADE) or JBees [3]. On the other hand, agents are considered useful for realizing flexible and adaptive workflows especially by using dynamic composition techniques based on semantic service descriptions, negotiations and planning techniques.

The discussion of related works shows that the complementary advantages of the different paradigms have led to a number of approaches that aim at combining ideas from different paradigms. Most of the approaches focus on a technical integration that achieves interoperability between implementations of different paradigms (e.g. FIPA agents and W3C web services). In contrast, this paper presented a unified conceptual model that combines the characteristics of services, components and agents.

[2] `http://jade.tilab.com`

6 Conclusions and Outlook

In this paper it has been argued that different classes of distributed systems exist that pose challenges with respect to distribution, concurrency, and non-functional properties for software development paradigms. Although, it is always possible to build distributed systems using the existing software paradigms, none of these offers a comprehensive worldview that fits for all these classes. Hence, developers are forced to choose among different options with different trade-offs and cannot follow a common guiding metaphor. From a comparison of existing paradigms the active component approach has been developed as an integrated worldview from component, service and agent orientation. The active component approach has been realized in the Jadex platform, which includes modeling and runtime tools for developing active component applications. The usefulness of active components has been further illustrated by an application from the disaster management domain.

As one important part of future work the enhanced support of non-functional properties for active components will be tackled. In this respect it will be analyzed if SCA concepts like wire properties (transactional, persistent) can be reused for active components. Furthermore, currently a company project in the area of data integration for business intelligence is set up, which will enable an evaluation of active components in a larger real-world setting.

References

1. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *CoopIS*, pages 1226–1242. Springer, 2003.
2. M. Dragone, D. Lillis, R. Collier, and G. O'Hare. Sosaa: A framework for integrating components & agents. In *Symp. on Applied Computing*. ACM Press, 2009.
3. L. Ehrler, M. Fleurke, M. Purvis, B. Tony, and R. Savarimuthu. AgentBased Workflow Management Systems. *Inf Syst E-Bus Manage*, 4(1):5–23, 2005.
4. P. Jezek, T. Bures, and P. Hnetynka. Supporting real-life applications in hierarchical component systems. In *7th ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA 2009)*, pages 107–118. Springer, 2009.
5. R. Krutisch, P. Meier, and M. Wirsing. The agent component approach, combining agents, and components. In *1st German Conf. on Multiagent System Technologies (MATES)*, pages 1–12. Springer, 2003.
6. J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.
7. B. Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, 1993.
8. A. Pokahr and L. Braubach. Active Components: A Software Paradigm for Distributed Systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*. IEEE Computer Society, 2011.
9. A. Pokahr, L. Braubach, and K. Jander. Unifying Agent and Component Concepts - Jadex Active Components. In *MATES'10*. Springer, 2010.
10. M. Singh and M. Huhns. *Service-Oriented Computing. Semantics, Processes, Agents.* Wiley, 2005.
11. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.