Towards a Decentralized Middleware for Composition of Resource-limited Components to Realize Distributed Applications

Christian Bartelt, Benjamin Fischer and Andreas Rausch
Department of Computer Science, University of Clausthal, Clausthal-Zellerfeld, Germany

Keywords: Component Composition, Self Adaptation.

Abstract: Dynamic adaptive middleware solutions for component-based development have become very important for creating complex applications in recent years. Many different middleware systems have been developed. In addition, decentralized middleware systems have been developed for special areas such as ambient intelligence or generic middleware systems for a wide range of areas. However no decentralized middleware system based on composition of limited components has been constructed. No component can be connected to an unlimited set of other components, because every connection uses a small amount of resources like network traffic or processor time. Specifically in mobile system resources is very restricted. Therefore, we need a middleware to solve the competition for the needed components to get a good composition. This paper demonstrates an approach towards a procedure to compose components under the aspect of limited components. It also gives users the opportunity to prioritize an application to prefer it while creating a composition.

1 INTRODUCTION

In recent years, the application areas of complex software systems which can react to a changing environment have distinctly increased. Component-based software and systems development is the main foundation through which complexity and dynamic adaptability is handled.

To support the development of dynamic adaptable systems, several middlewares are provided (Klus et al., 2007); (Clarke et al., 2001). These middlewares can compose components at run time. Hence, developers of components can concentrate on the correctness of their components, rather than how to connect them. There are many different middlewares for different environments. For example, in the field of ambient intelligence (Issarny et al., 2004) and in the field of ubiquitous computing (Kon et al., 2002).

However none of these middlewares deal with the fact that a service of a component can only be used for a restricted number of components.

Most middlewares allow an optimal composition of components for example DAiSI (Klus et al., 2007) or OpenCOM (Clarke et al., 2001). Therefore, an optimal composition is typically determined by one central unit with global knowledge about all potentially connectable components.

On the one hand, algorithms often need a long time to determine the best composition for a large set of components. On the other hand, in many cases a central unit cannot be used. This is because there is not enough calculation power to realize a central unit or, in other cases, every component or unit can be missing in the system. For this reason, some middleware systems use a decentralized approach to compose (Baresi et al., 2008). Of course there are middleware systems which consider restricted resources. However their solution for that problem is a small middleware which uses only a small amount of the resources such as calculation power (Janakiram et al., 2005). What should be done, if this isn't enough, to realize the necessary set of components?

The goal of our work is to construct a dynamic adaptive middleware which connects components without a central unit which takes note of the restriction of the services of individual components. The next chapter deals with an issue, where such a middleware is needed and the challenges that the middleware must deal with. Chapter 3 clarifies an abstract example and creates a formal description in order to get a general comprehension of components and to describe what exactly qualifies a good

Bartelt C., Fischer B. and Rausch A..
Towards a Decentralized Middleware for Composition of Resource-limited Components to Realize Distributed Applications.
DOI: 10.5220/0004386402450251
In *Proceedings of the 3rd International Conference on Pervasive Embedded Computing and Communication Systems* (SANES-2013), pages 245-251
ISBN: 978-989-8565-43-3
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

connection between components. In Chapter 4, the formal description introduced in Chapter 3 is extended and our approach is clarified. Chapter 5 contains the overall summary and further work.

2 PROBLEM

The following example demonstrates the necessity of such a decentralized middleware in the field of disaster management. In case of emergency aid, many groups are involved to provide help. Every group has its own specific function. There are coordinators for the other groups; auxiliaries, for example, specialists, who search whether a place is safe or there is an area where an explosion could occur; etc. In order to maximize the efficiency, a communication infrastructure must be built. The easiest way to do is as follows: Headquarters builds the communication infrastructure and coordinates all information. The coordination belongs to the coordinators, who are often near headquarters. However headquarters is not always the first to reach a place of an emergency. Hence, the other groups should communicate without headquarters in addition. In large-scale disasters, the information headquarters has to handle is too much. It is therefore useful to use more headquarters to collect and filter the information before it is forwarded. In order to assist these groups, a communication system should be built. Although there is no central group which is always the first to reach a disaster, the communication system must function well. The communication system must also realize that the number of information a single group can receive is restricted, for example in using multiple headquarters (see Figure 1).

A good communication system allows all auxiliaries to send their information to at least one coordinator.

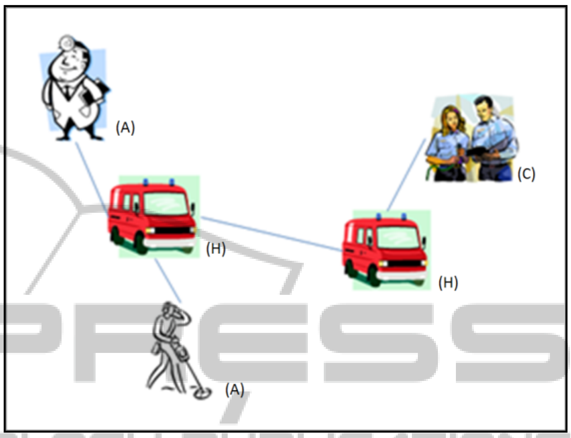


Figure 1: A communication system for disaster management containing auxiliaries (A), headquarters (H) and a coordinator (C).

The following example shows the approach of such a system. This example contains 3 types of units: coordinators, who coordinate the information, auxiliaries, who collect the information and fulfil orders, and headquarters, which filters and forwards information. For technical support, each of the auxiliaries and coordinators uses a PDA.

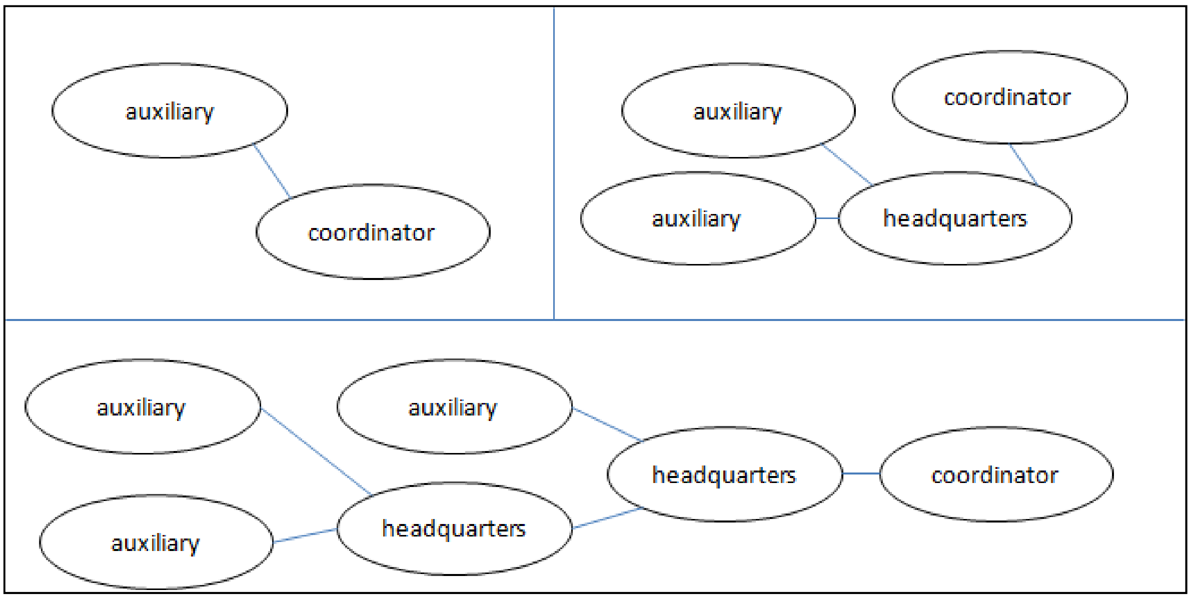


Figure 2: Examples of a communication infrastructure.

In this example, an auxiliary and a coordinator can communicate to only one other unit. A headquarters can establish a communication between one coordinator or one other headquarters and every combination of two units, which can be headquarters or auxiliaries, is described in Figure 2. As described before, to realize a good composition of a communication infrastructure, all auxiliaries have to be connected.

2.1 Challenge

The goal of such a communication system is that every auxiliary can send his information to at least one coordinator. Hence, all auxiliaries must have an implicit connection to a coordinator. This system must consist of the two aspects described above. Firstly it must be built without a central unit. Therefore, a central algorithm cannot be used. The other problem to deal with is that every unit can only communicate to a restricted number of other units.

3 ABSTRACTION

In order to define a formal description, an abstraction of the example introduced in Chapter 2 has been made (see Figure 3). All groups will be declared as components and all units as instances of components. A group of connected instances called configuration defines an application.

There is no fixed description of components. The description widely accepted and used is that of Clemens Szyperski:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. "(Szyperski et al., 2002).

In this abstract example, there are 4 components which use or implement the two Interfaces "A" and "B". Five instances of the four components have been created and two possible configurations of these instances have been made. The quality of a configuration depends on a set of components, which do not implement an Interface and the whole chain of instances, which they are connected to, have all required interface connectors connected. The configuration at the top of the diagram is the best configuration, because the two important instances "a" and "b" are connected and the chain has no required connectors, which are free. The configuration at the bottom of the diagram is less effective, because the chain containing the instance "a" has one required connector, which is free. Therefore, this chain cannot run smoothly. Instances such as "a" and "b" will be called initial Instances.
In the following part of this chapter, a formal description of these units is made.

First some basic units must be defined:

T: set of all interface instances (1)

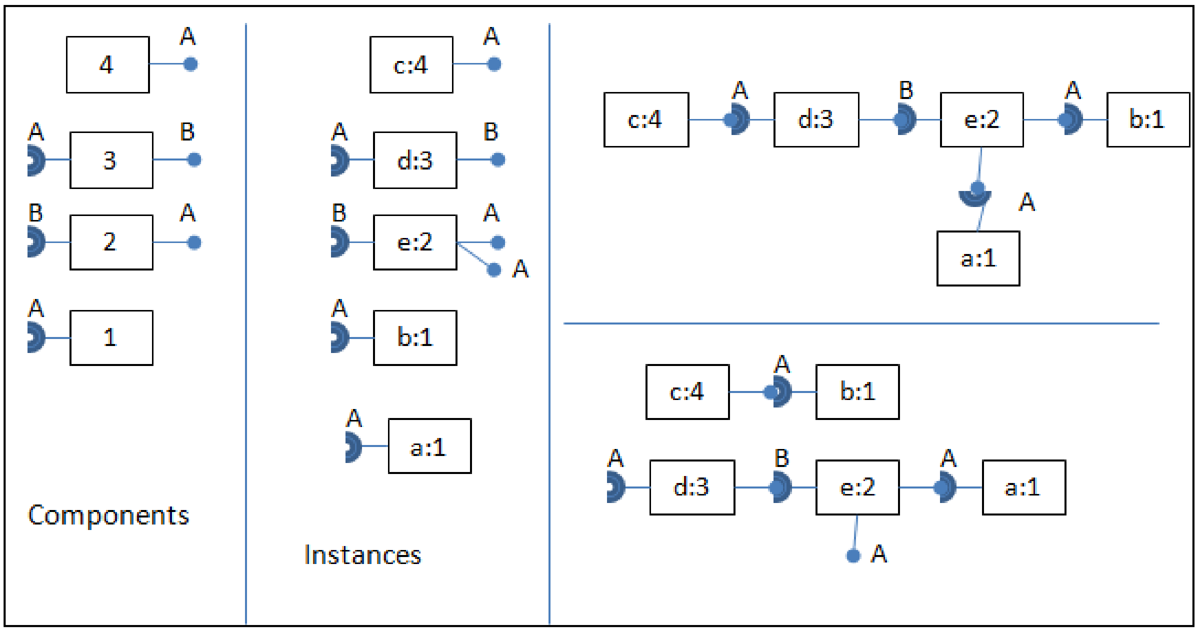
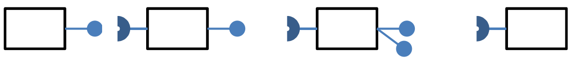


Figure 3: It illustrate components, instances of components and two possible compositions of these instances.

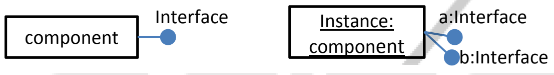
$$I = P(T) \quad P = \text{Partition} \tag{2}$$

$$C = \wp(I) \times \wp(I): \forall (r, p) \in C: r \cap p = \varnothing \text{ and } r, p \text{ are finite} \tag{3}$$



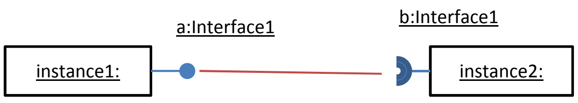
A component (C) contains a set of interfaces (I) which it implements (p) and knows a set of interfaces other components implement and this component can use to communicate to the other components (r).
To every component you can create instances (J).

$$J = \wp(T) \times \wp(T): \forall (r, p) \in J: r \cap p = \varnothing \tag{4}$$



An instance contains a set of providing interface instances (p) and a set of requiring interface instances (r). In order to connect these instances, for a complete communication system, the connection (S) was declared as the following.

$$S = \wp(T)\ \forall s \in S: |s|=2 \wedge \exists i \in I: \quad s = \{t_1, t_2\} \wedge t_1 \neq t_2 \wedge t_1, t_2 \in i \wedge \exists! j_1=(p_1, r_1), \exists! j_2=(p_2, r_2) \in J: t_1 \in p_1 \wedge t_2 \in r_2 \tag{5}$$

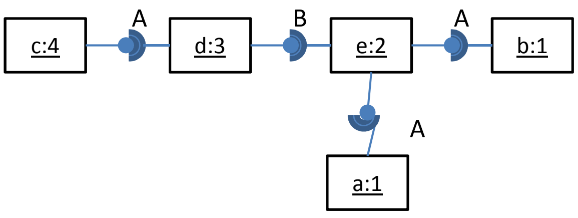


Every connection connects exactly two instances of components over the same interface. The one component implements that interface and the other component uses that interface.

The instances and connections together demonstrate that every instance can only be used by a limited set of other instances. This is necessary to describe the limitation in using instances of components, which is the main argument of this paper.

The next declaration describes an example communication system. Every possible composition will be called configuration (K):

$$K = \wp(J) \times \wp(S): \forall (j, s) \in K: \forall \{t_1, t_2\} \in s: \exists! (r_1, p_1), \exists! (r_2, p_2) \in j: t_1 \in r_1 \wedge t_2 \in p_2 \tag{6}$$



A configuration contains a set of instances and a set of connections between these instances. We are not considering cyclic dependencies within configurations. This will be done in further works. Therefore, a configuration has a hierarchic structure.

A function is needed to know the type of an instance.

$$\text{type}: J \rightarrow C$$

$$\text{type}(j) = c: j = (r_1, p_1) \wedge c = (r_2, p_2) \Rightarrow (\forall t \in r_1: \exists I \in r_2: t \in I) \wedge (\forall I \in r_2: \exists t \in I: t \in r_1) \wedge (\forall t \in p_1: \exists I \in p_2: t \in I) \wedge (\forall I \in p_2: \exists t \in I: t \in p_1) \tag{7}$$

Every instance belongs to just one component, according to the providing and requiring interfaces.

4 APPROACH

In order to facilitate the description of the approach further definitions are necessary:

$$\text{free}: J \times K \rightarrow \wp(T)$$

$$\text{free}(j, k) = \{t \mid j = (r, p) \wedge k = (j_2, s) \wedge t \in r \wedge \{t_1, t_2\} \in s \Rightarrow t \neq t_1 \wedge t \neq t_2\} \tag{8}$$

$$\text{bound}: J \times K \rightarrow \wp(J)$$

$$\text{bound}(j, k) = \{j2 \mid k = (j1, s) \wedge j = (r1, p1) \wedge j2 = (r2, p2) \in j1 \Rightarrow \exists t1 \in p2, \exists t2 \in r1: \{t1, t2\} \in s\} \tag{9}$$

These two functions show whether the instance has requiring connectors which are not connected to another instance and the instances which are connected to the requested instance.

$$\text{active}: J \times K \rightarrow \text{BOOL}$$

$$\text{active}(j, k) = \begin{cases} 1 & \text{if } |\text{free}(j, k)|=0 \wedge \forall j_2 \in \text{bound}(j, k): \text{active}(j_2, k) \\ 0 & \text{else} \end{cases} \tag{10}$$

An instance will be active if two criteria are successful. All requiring connectors of the requested instance are connected to other instances and all those instances are themselves active. Although this function is recursive, it will terminate every time, because of not having cycles in configurations.
The function to describe the assessment of a communication system is given below.

$$\text{assess}: K \rightarrow \text{IN}$$

$$\text{assess}(k) = |\{i_2 \mid k = (i_1, s) \wedge i_2 \in i_1 \Rightarrow (r, p) = \text{type}(i_2) \wedge |p|=0 \wedge \text{active}(i_2, k)\}| \tag{11}$$

It simply counts all active instances whose components have no providing interfaces. In other words, it counts all active initial instances.

For the composition of instances we need two more functions.

$$\text{bind: } J \times J \times K \rightarrow K$$

$$\text{bind}(j_1, j_2, k_1) = k_2 : k_1 = (j_3, s_1) \wedge k_2 = (j_4, s_2) \Rightarrow j_3 = j_4 \wedge s_1 \subseteq s_2 \wedge \exists t \in s_2 : t = \{j_1, j_2\} \quad (12)$$

$$\text{release: } J \times J \times K \rightarrow K$$

$$\text{release}(j_1, j_2, k_1) = k_2 : k_1 = (j_3, s_1) \wedge k_2 = (j_4, s_2) \Rightarrow j_3 = j_4 \wedge s_2 \subseteq s_1 \wedge (\forall s \in s_2 : s \neq \{j_1, j_2\}) \wedge \exists s \in s_1 : s_2 \cup \{s\} = s_1 \quad (13)$$

With these functions we could create a random configuration. However, our abstract example demonstrates that some configurations are better than other. The best configuration is one, where as many initial instances as possible are active.

Our approach extends the definition of instance (J):

$$J = \wp(T) \times \wp(T) \times \mathbb{N} : \forall (r, p, n) \in J : r \cap p = \varnothing \quad (14)$$

Now, the instance contains a number, which belongs to the priority of the initial instance. The user can set a priority to the initial instance and therefore to the application which it starts. An application with a higher priority has a higher chance of running all needed instances than an application with a lower priority. In the following text, the algorithm creating a good configuration by using this system of priority will be shown.

4.1 Composition Procedure

In this subsection an algorithm is described, which should compose the instances into a good configuration. A decentralized system has to be built. Therefore all instances use the algorithm on their own to create a good configuration for all initial instances.

The algorithm is structured in steps, as follows:

1. All initial instances and instances which have a request for implemented interfaces and need other instances themselves broadcast a request. The request contains the interfaces it is searching for and a priority of the searching instances. Only the initial instances get a priority from the user. Hence, the other instances can calculate their own priority (see step 2).
2. Every instance which can respond to a request notices the request. It will now calculate its priority.
   a. For every connector the instance has providing it, it will search in the list of calls the one who matches to the connector and notice the priority if a matched request exists.
   b. This request will be ignored for the other connectors, because every request has to be used for just one connector at a time.
   c. Finally, all noticed priorities are added together and create the new priority for the instance.

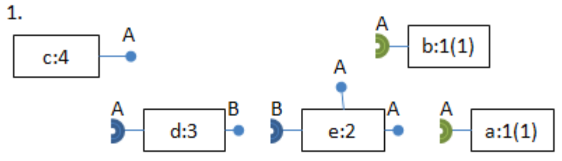
   If the set of all instances does not change between two calculation steps, every priority will be const.
3. If an instance which can respond to a request doesn't need instances itself or if it has all needed connectors connected, it will respond to the request with the highest priority. If the response is denied, the instance responds to the next matched request with a lower priority. If it already has this connector connected and the new request has a higher priority than the connected instance, it will free the connector and will respond to the new request.
4. If an instance responds to its request, it accepts only the amount of response it needs and denies the others. Then it connects to the instances which have responded the request.
5. If every required connector is connected, the initial instance can start running.

The following subsection shows the algorithm for the abstract example. Note that all components run parallel. In this example, all initial instances which can run are running.

4.2 Example

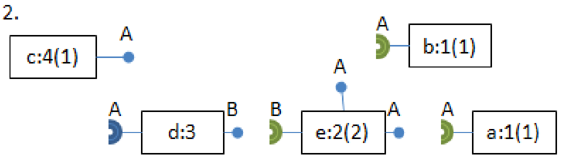
This subsection illustrates the algorithm defined in the example described in Chapter 3.

1.

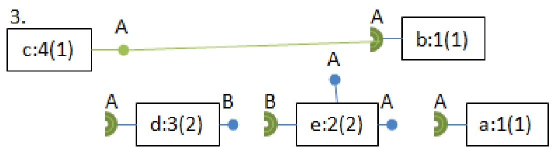


In Step 1 both initial instances "a" and "b" broadcast a request. In this case, the interface "A" is searched.

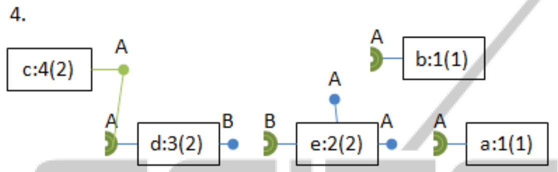
2.



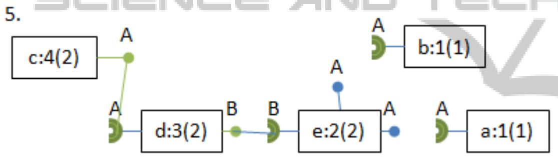
In Step 2 instances "c" and "e" calculate their own priority and instance "e" broadcasts a request itself.



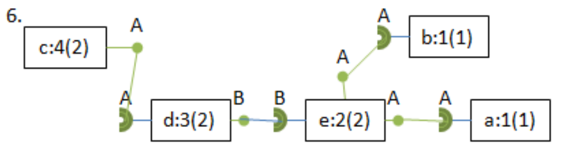
Now instance "c" can respond to the calls of instances "a" and "b". As they had the same priority he chose one of them at random. Instance "d" calculates its priority and broadcasts a request.



Instance "c" notices an instance requested for interface "A" with a higher priority than it is connected, so it disconnects from instance "b" and connects to instance "d".



In Step 5, instance "d" has all required connectors connected and can respond to the request of instance "e" and can connect to "e".



Finally instance "e" has all required connectors connected and connects to "a" and "b". Because no instance can respond to a request with a higher priority the system reaches a stable state.

4.3 Result

As illustrated above, the algorithm creates the best configuration for this example as described in Chapter 3. Therefore, it could be a good approach to solve problems such as this.

The algorithm uses a broadcast to let every instance know the requests. It is similar to an algorithm using a central unit for composing, but there are also differences. To composite components a central unit needs more information about the components, than information broadcasted in the requests. For example the central unit need to know which component provides which interface. Another difference is, that every instance can only remember those requests, it can provide. Therefore, every instance has less information, than a central unit.

In order to project the abstract example onto the example described in Chapter 2, an auxiliary should be defined as an initial instance. Therefore, the algorithm tries to connect all auxiliaries to at least one coordinator. This implies that all information of the auxiliaries could be sent to at least one coordinator.

5 SUMMARY AND FURTHER WORK

In this paper, an approach which can connect a number of instances together without a central unit by having regard to the restriction of the services one instance can serve has been demonstrated. A real-life example has been shown where middleware could be used, for example, to build a communication infrastructure. In further work, this approach will be evaluated and extended with further properties. One property of components in most systems is that not every required Interface is necessary to run the instance of the component. Hence, the approach needs to take this into account.

REFERENCES

Kon, F., Costa, F., Blair, G, Campbell, R. H., 2002. *The Case for Reflective Middleware*. CACM June 2002/Vol. 46, No. 6.

Issarny, V., Sacchetti, D., Tartanoglu, F., 2004. Developing ambient intelligence systems: A solution based on web services. *Journal of Automated Software Engineering*.

Szyperski, C., Gruntz, D., Murer, S., 2002. The book, Component Software: *Beyond Object-Oriented Programming*. New York, 2nd Edition, Addison-Wesley.

Currion, P., Silva, C., Van de Walle, B., 2007. Open source software for disaster management. *Communications of The ACM*, Vol. 50, Issue 3, pp.61-65.

Klus, H., Niebuhr, D., Rausch. A., 2007. A component model for dynamic adaptive systems. In Alexander L. Wolf, editor, *Proceedings of the International Workshop on Engineering of software services for pervasive environments*, pages 21–28, Dubrovnik, Croatia.

Janakiram, D., Venkateswarlu, R., Nitin, S., 2005, COMiS: Component Oriented Middleware for Sensor Networks. To appear *in the proceedings of 14th IEEE Workshop on Local Area and Metropolitan Networks*

(LANMAN), Chania, Crete, Greece.

Clarke, M., Blair, G. S., Coulson, G., Parlavantzas, N., 2001. *An efficient component model for the construction of adaptive middleware*. In Middleware, Springer-Verlag, pp. 160–178.

Baresi, L., Guinea, S., Tamburrelli, G., 2008. Towards decentralized self-adaptive component-based systems. *In Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 57.