Adaptive Component Allocation in ScudWare Middleware for Ubiquitous Computing

Qing Wu and Zhaohui Wu

College of Computer Science, Zhejiang University,
Hangzhou, Zhejiang, China, 310027
{wwwsin, wzh}@cs.zju.edu.cn

Abstract. With the increasing prevalence of ubiquitous computing, the software component allocation while meeting various resources constraints and component interdependence is crucial, which poses many kinds of challenges. This paper mainly presents an adaptive component allocation algorithm in ScudWare middleware for ubiquitous computing, which uses dynamic programming and forward checking methods. We have applied this algorithm to a mobile music space program and made many experiments to test its performance. The contribution of our work is twofold. First, our algorithm considers resources constraints requirement, component interdependence, and component tolerant issues. Second, we put forward a component interdependence graph to describe interdependent relationships between components. As a result, the evaluation of component allocations has showed our method is applicable and scalable.

1 Introduction

In recent years, computations are becoming ubiquitous and embedded[1], which provide more facilities for our life. Ubiquitous computing aims at fusing physical world and information space naturally and seamlessly, which demands plenty of computation resources for performance requirements. It must require considering interdependences of functional aspects. However, the computation resources in ubiquitous and embedded environments are limited such as CPU computation capabilities, network bandwidth, and memory size. As a result, it sometime cannot provide enough resources to execute some applications successfully. In addition, changes of the heterogeneous contexts including people, computing devices and environments are ubiquitous. Therefore, it results in many problems in software design and development. We think adaptation is the key issue of software systems and applications to meet the different computing environments.

In this work, we focus on the design-time adaptation of component allocation considering resources constraints, component interdependence, and tolerant issues. This paper presents a scalable algorithm using dynamic programming and forward checking methods to allocate components meeting above three aspects in design-time. We have implemented this method for ScudWare middleware, applied it to a mobile music space program, and made many experiments to

evaluate its performance. The result shows the method is applicable and scalable.

The rest of the paper is organized as follows. Section 2 presents a ScudWare middleware platform. Section 3 details a computation model consisting of the adaptive component formalization and a component interdependence graph. Section 4 proposes an adaptive component allocation algorithm. In section 5 and 6, we give a case study and make experiments to evaluate performance and scalability of the algorithm. Then some related work is stated in section 7. Finally, we draw a conclusion in section 8.

2 ScudWare Middleware Platform

We have developed a ScudWare[2] middleware platform in terms of CCM (CORBA Component Model)[3] specification, using ACE (Adaptive Communication Environment)[4] and TAO (The ACE ORB)[5]. TAO is a real-time ORB (Object Request Broker) developed by Washington University.

The ScudWare architecture consists of three parts: ACE components, real-time ORB core and ScudCCM, defined as SCDW=(ACE, ETAO, SCUDCCM). (1) ACE denotes an adaptive communication environment and is targeted for developers of high performance and real-time communication services. (2) ETAO, the extended ACE ORB, is a CORBA middleware framework that allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, and communication protocols. ETAO includes a set of services such as transaction, persistence, and life cycle services. In addition, we have developed context-aware, adaptive notification, and component management services added to ETAO. (3) SCUDCCM=(CC, CH, ACM), where CC denotes a set of component containers, which are environments of component runtime; CH denotes a set of component homes, managing component lifetime; ACM denotes adaptive component management. It includes component package, assembly, deploy, and allocation in design-time. Besides, it consists of component migration, replacement, updating, and variation in runtime. Overall, in this architecture, we emphasize on context-aware characters and adaptive mechanisms by providing context-aware service, adaptive service, and component management.

3 Computation Model

In this section, we define an adaptive component computation model in a structural method and describe the problems while meeting resource constraints and component interdependence during the component allocation. We firstly give some definitions about the adaptive component. Then we introduce a component interdependence graph.

3.1 Adaptive Component Formalization

In turn, we give definitions of adaptive component model, adaptive component organization model, adaptive component container model, and adaptive component community model.

Definition 1 *An adaptive component model* $A_C = (C, I, O, R, P)$ is a meta-object. (1) C denotes component's capabilities, including a set of computation functions. (2) I denotes a set of input interfaces provided by other components. (3) O denotes a set of interfaces exporting for other components use. (4) R denotes a set of required resources consumptions value (e.g. CPU computation, network communication bandwidth, and memory size). (5) P is the executing platform of component(e.g. Windows and Linux). Specially, if two components (e.g.a_c^1 and a_c^2) have the same interfaces and capabilities and run at the same platform, we call them component brothers, which is defined as $brother(a_c^1) = a_c^2$. Besides, if two components (e.g. a_c^1 and a_c^2) are not brothers, but having same capabilities regardless of other aspects (e.g. platform feature, interfaces), we call them component friends defined as $friend(a_c^1) = a_c^2$.

Definition 2 *An adaptive component organization model* $A_o = (G, IL, OL, F, RC)$ is a weighed directed component graph. (1) $G \subseteq A_c$ denotes a group of components. (2) IL denotes a set of directed links from the output interface of one component to the input interface of another component. (3) OL denotes a set of directed links from the input interface of one component to the output interface of another component. (4) F specifies the semantic description of IL and OL. (5) RC denotes a set of resource consumption functions as:

(a) *CPU Computation Consumption*: $RC_{cc} : \forall c \in A_c \cdot \exists v \in Q^+ \cdot (RC_{cc} \rightarrow v)$ defines the CPU computation resource consumption by component c. Q^+ is a set of non-negative real numbers.

(b) *Communication Consumption*: $RC_{cm} : \forall c \in A_c \cdot \exists v \in Q^+ \cdot (\sum RC_{cm} \rightarrow v)$ defines communication resource consumption by component c.

(c) *Memory Consumption*: $RC_{mm} : \forall c \in A_c \cdot \exists v \in Q^+ \cdot (RC_{mm} \rightarrow v)$ defines memory resource consumption by component c.

According to above definitions, we have modelled the component resources consumptions as a directed graph. The node weight denotes the CPU computation and memory consumption. In addition, the node links' weight denotes component communications' consumption.

Definition 3 *An adaptive component container model* $A_n = (P, RP, RA, CA, D)$ is a weighed undirected graph. (1) P denotes a set of execution platforms including different OSs and middleware infrastructures. (2) $RP = (RP_{cc}, RP_{cm}, RP_{mm})$ is a set of current left available resources consisting of CPU computation, network bandwidth, and memory resources. (3) $RA = (RA_{cc}, RA_{cm}, RA_{mm})$ is a set of resources that have been allocated for CA. (4) $CA \subseteq A_c$ is a set of components allocated into P. The definitions of RP and RA are same as RC in definition 2. (5) D denotes a set of containers' lifetimes.

Definition 4 *An adaptive component community model* $A_m = (A_o, A_n, K, CT)$ is a component network permeated in ubiquitous computing environments. (1) A_o denotes a set of adaptive component organizations including many het-

erogeneous components. (2) A_n denotes a set of various adaptive component containers. (3) K is a set of human-centric tasks. (4) CT is a set of current context information related to application domains.

In terms of above four definitions, we can conclude the component allocation problem formally equals to a model-building problem. That means how to build *an adaptive component community model* A_m by giving *an adaptive component organization model* A_o and *an adaptive component container model* A_n. Specifically stated, there are three restrictions:

(a) Each component is allowed to be allocated in just one container during its lifetime.

(b) It must meet restrictions of resources quantity. That means RP should be more than RA in A_n.

(c) During component allocation, we should consider component interdependence. For example, if component c_1 *has been allocated into container* n_1 and component c_1 and c_2 are interdependent directly, c_2 must be allocated in container n_1 regardless of other aspects.

3.2 Component Interdependence Graph

In order to describe the interdependent relationships between components, we introduce a component interdependence graph composed of component nodes and link paths.

For each component, we associate a node. The link paths are labelled with a weight. We define a component interdependence graph $A_{ig} = (CN, LP, W)$. (1) $CN = \{cn_i\}_{i=1..n}$ denotes a set of component nodes. (2) $LP = \{l_{i,j}\}_{i=1..n,j=1..m}$ denotes a set of component links, describing the dependent targets. $l_{i,j}$ is the link between the nodes cn_i and cn_j. (3) $W = \{w_{i,j}\}_{i=1..n,j=1..m}$ denotes a set of interdependent weight. w_{ij} is a non-negative real number, which labels $l_{i,j}$. In addition, $w_{i,j}$ reflects the importance of the interdependence between the associated components. These weights used, for instance, to detect which links becomes too heavy or if the systems rely too much on some components. In terms of this weight, we can decide which component should be allocated preferentially. Extremely, this graph changes in terms of the different contexts. Therefore, this interdependence is not static: it can be modified when a new component is added or one component disappears. Moreover, based on the different application domain context and run-time environment, the interdependent relationships will change.

4 Adaptive Component Allocation Algorithm

This section describes an adaptive component allocation algorithm, considering resources constraints requirement, component interdependence, and component tolerant issues, which we call *RIT-Based component allocation algorithm*. Besides, this algorithm uses *DP* (*dynamic programming*) and *FC* (*forward checking*) methods, which invokes functions *CCAP*, *SCB* and *ACDC*.

4.1 RIT-Based Component Allocation Algorithm

Giving a set of application domains and human-centric tasks, *RIT-Based algorithm*, listed in algorithm 4.1, aims at generating *an adaptive component community model* A_m from *an adaptive component organization model* A_o, *an adaptive component container model* A_n, and a set of *human-centric tasks* K in terms of resources constraints requirement, component interdependence, and component tolerant issues. First, *RIT-Based algorithm* decomposes tasks into a set of logic functions. Second, it finds a set of key components and their *brothers* and *friends*. Third, it forms the *component interdependence graph*. Fourthly, for each component, it calculates its *allocation priority* according to the resources quantity required and the interdependent relationships between other components. Then it will sort components in terms of their allocation priorities. Specially stated, for each key component, its one brother or friend will be selected as a component backup for tolerant issues. Finally, it completes all component allocations and returns the allocation scheme.

Algorithm 4.1 *RIT-Based Component Allocation*

Input: (1) *An adaptive component organization model* A_o *=(G, IL, OL, F, RC)*; (2) *An adaptive component container model* A_n*=(P, RP, RA, CA, D)*; (3) *A set of human-centric tasks* K*=*$\{lf_1, lf_2, ..., lf_n\}$.

Output: *An adaptive component community model* $A_m = (A_o, A_n,$ *K, CT).*

Begin

Step 1 : *Decomposing* K *into a set of logic functions* $\{lf_1, lf_2, ..., lf_n\}$;

Step 2 : *Finding a set of key components and their brothers and friends in terms of* lf_i;

Step 3 : *Forming the component interdependence graph*;

Step 4 : (1) CCAP : *Calculating all components' allocation priorities and sorting them*; (2) SCB : *Selecting component backups for some key components*;

Step 5 : (1) ACDC: *Allocating all components into different component containers, and generating* A_m; (2) *If* $(A_m.A_o.G = A_m.A_n.CA)$ *then return* A_m *else return error.*

End.

4.2 CCAP and SCB

For *CCAP*, we introduce an adaptive competence function for each component. We consider both combined resources consumptions of each component and its interdependence. First, for each component $c \in A_o.G$, the component allocation priority *CCAP(c)* is calculated as formula (1).

$$(\alpha*(\frac{RC_{cc}(c)}{\sum_{i=1}^{n} RP_{cc}(A_n(i))}) + \beta*(\frac{RC_{mm}(c)}{\sum_{i=1}^{n} RP_{mm}(A_n(i))}) + \gamma*(\frac{\sum RC_{cm}(c)}{\sum_{i=1}^{n} RP_{cm}(A_n(i))}))*(1+\sum w(c)) \tag{1}$$

We use α, β and γ to define the weights of CPU Computation, communication and memory resources. If we emphasize CPU Computation more than other aspects, we can set α more than β and γ. In addition, $(1+\sum \mathrm{w(c)}$ considers the aspect of component interdependence.

In order to add tolerant mechanism, we introduce the component backup method for key components. In *SCB* step, we select one component brother or friend as its backup. For example, if c_1 is one of key components and has brother c_2, c_2 will be as its backup. If c_1 has not any brothers, but has friend c_3, c_3 will act as its backup. Emphatically, the key component and its backup must be allocated into the different component containers at the same time. For instance, if there are key component c_1 and its backup c_2, and c_1 is placed into component container cn_1, c_2 should be allocated into another component container except cn_1.

4.3 ACDC

For the allocation of each component, we use *DP*(*dynamic programming*) and *FC*(*forward checking*) methods. In *ACDC*, we present three allocation policies:

(a) *If component cn_1 and cn_2 are directly interdependent, we must allocate them into the same component container.*

(b) *Assume that there are two component containers: cn_1 and cn_2. Component c_1 will be placed. If c_1 is to be placed into cn_1, $x_1 = \sum(cn_1.RA/cn_1.RP)$. Also, if c_1 is to be placed into cn_2, $x_2 = \sum(cn_2.RA/cn_2.RP)$. As a result, if $x_1 \leq x_2$, we will allocate c_1 into cn_1, else c_1 will be placed into cn_2.*

(c) *If the allocation of component c_1 dose not meet the resource constraint requirement according to (1) and (2), c_1 will be placed into the component container that has the maximum available resources.*

According to above three policies, we use *DP* and *FC* methods to decide where to place each component, which reduce component allocation complexities.

5 Case Study

In order to demonstrate the usefulness and effectiveness of our component allocation method, we apply algorithm 4.1 to a MMS (mobile music space) program[6] in smart vehicle space. MMS program aims at acquiring music source, playing music, and outputting music adaptively in terms of diverse context information.

The structural model of MMS consists of components for *MSM* (*music source management*), *PM* (*playing music*), *OM* (*outputting music*), *CM* (*context management*), *CA* (*context acquisition*), *CR* (*context reasoning*), *DF* (*detecting fault*) and *CC* (*Controlling center*). Figure 1 shows this structural and resources consumptions of each component and their interdependent weight (Kilobytes for memory and links). There are two component containers: a_n^1 and a_n^2, which are connected via a shared link. Their available resources include 20 and 25 computation, 30 KB and 35 KB memory, and 100 KB/s communication link. As following, we will illustrate the steps of component allocations.

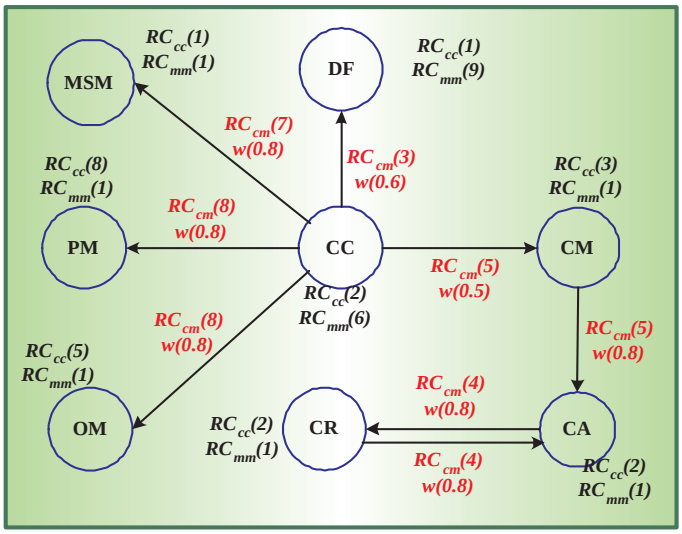

Fig. 1. MMS Component Resources Consumptions and Interdependence Graph

First, we calculate allocation priorities of all components and sort components by formula (1). Specially stated, we assign $\alpha = \beta = \gamma = 1$. Based on the result of calculation, the order of component allocated is (*CC, CM, PM, OM, CA, CR, MSM, DF*), where CCAP(*CC*)=2.0102, CCAP (*PM*)=0.2732, CCAP (*CM*)=0.2377, CCAP (*OM*)=0.2065, CCAP (*DF*)=0.1907, CCAP (*CA*)=0.1796, CCAP (*CR*)=0.1796, and CCAP (*MSM*)=0.1076.

Next, we allocate above components into two containers according to three policies of *ACDC*. Because CCAP(*CC*) is maximum, *CC* is the key component. We select its one brother or friend as a backup to allocate to another container at the same time. In addition, because *CA* and *CR* are interdependent directly, they must be placed into the same component container. Table 1 shows the component allocation steps and related values. In this case, forward checking is not performed because the allocation of each step meets resources constraints. Emphatically stated, if component c is calculated to placed into a_n^1, however a_n^1 has not enough resources, forward checking is done and component c will be allocated to a_n^2.

Finally, component *CC, PM, CM, DF*, and *MSM* are placed into component container a_n^2, and component *CC-backup, OM, CA,* and *CR* are allocated into component container a_n^1.

6 Experiments Evaluation

In order to test the performance and scalability of above adaptive component allocation algorithm, we have made many experiments. The evaluation uses the number of visited components as a metric for measuring algorithm efficiency. We

Table 1. Component Allocation Steps

Steps	Components	X(a_n^1)and X(a_n^2)	Allocation
1	CC(2.0102)	0.3000/0.2514	a_n^2 (CC-backup→ a_n^1)
2	PM(0.2732)	0.5657/0.3823	a_n^2
3	CM(0.2377)	0.2583/0.2357	a_n^2
4	OM(0.2065)	0.3995/0.5596	a_n^1
5	DF(0.1907)	0.1490/0.1370	a_n^2
6	CA(0.1796)	0.2424/0.2607	a_n^1
7	CR(0.1796)		a_n^1 (ACDC: policy (1))
8	MSM(0.1076)	0.1735/0.1496	a_n^2

consider varying the number of both components and containers. For comparison, we choose *SDP* (*standard dynamic programming*) as a baseline, compared to our method *DPFC* (*dynamic programming with forward checking*). The experiments conclude our algorithm has good performance and scalability.

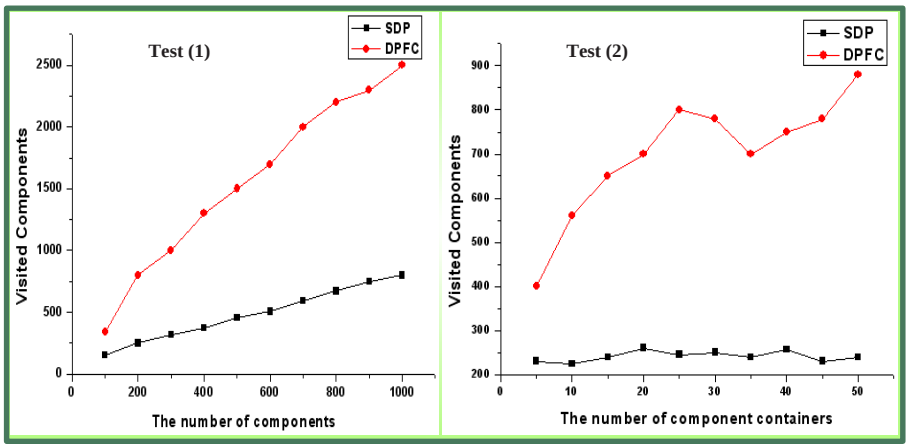

Fig. 2. The numbers of visited components in Test (1) and Test (2)

Test 1: Varying the number of components. We evaluate the scalability of the algorithm varying the number of components. First, we fix the number of component containers to be 6, which have different number of resources. Second, we select the number of components from 100 to 1000 in increments of 100, which resource consumptions are randomly generated. Third, we use *SDP* and *DPFC* algorithms respectively to count the number of visited component once a solution found. The result is shown in figure 2. The numbers of visited com-

ponents caused by *SDP* and *DPFC* both increase as the number of components increases. However, if the number of components is same, the numbers of visited component caused by *SDP* and *DPFC* are different significantly, which is due to forward checking method that improves the solution search.

Test 2: Varying the number of component containers. We evaluate the scalability of the algorithm varying the number of component containers. First, we fix the number of components be 100, which resources consumptions are randomly generated. Second, we select the number of component containers from 5 to 50 in increments of 5, which have different number of resources. Third, we use *SDP* and *DPFC* algorithms respectively to count the number of visited component once a solution found. Figure 2 also shows experiment results. The numbers of visited components caused by *SDP* and *DPFC* both increase as the number of components increase. Nevertheless, on the same number of components, the numbers of visited component caused by *SDP* and *DPFC* are different significantly, which is also due to forward checking method.

7 Related Work

Software component adaptation has the potential for enhancing the system's flexibility and reliability to a very wide range of factors. Adaptive component allocation is playing an important part in software adaptation. Many efforts have been put in this research area.

Shige Wang and Kang G. Shin[7] give a method of component allocation using an informed branch-and-bound and forward checking mechanism subject to a combination of resource constraints. However, their method ignores the execution dependencies of components and tolerant issues when the structural model is partitioned. Belaramani and Cho Li Wang[8] propose one dynamic component composition approach for achieving functionality adaptation and demonstrate its feasibility via the facet model. Nevertheless, they do not consider related aspects in design-time fully. Philip K. Mckinley[9] considers the compositional adaptation enables software to modify its structure and behavior dynamically in response to changes in its execution environment and gives a review of current technology compares how, when, and where re-composition occurs. Kurt Wallnau and Judith Stafford[10] discuss and illustrate the fundamental affinity between software architecture and component technology. They mainly outline criteria for the component integration. Besides, we have proposed a semantic and adaptive middleware[11] and a component management framework for data management in smart vehicle space. According to us, we should consider resources constraints, component interdependence, and tolerant issues for adaptive component allocation in design-time adequately.

8 Conclusions and Future Work

Adaptive component allocation with many aspects' constraints is playing an important role for software component design in ubiquitous computing environ-

ments. This paper presents an adaptive component allocation algorithm applied to the ScudWare Middleware, considering resources constraints, component interdependence, and tolerant issues. This algorithm uses dynamic programming and forward checking methods. We have made many experiments to test the performance and scalability of the algorithm. The experiment results show our method has good performance and scalability.

Our future work aims at improving the performance of our component allocation algorithms. In addition, we will take other methods to realize more flexibility and reliability of component allocation in design-time.

Acknowledgments

This research was supported by 863 National High Technology Program under Grant No. 2003AA1Z2080, 2003AA1Z2140 and 2002AA1Z2308.

References

1. Weiser M: The Computer for the 21st Century. Scientific American, pp. 94-100 (1991)
2. Zhaohui Wu, Qing Wu, Jie Sun, Zhigang Gao, Bin Wu, and Mingde Zhao: ScudWare: A Context-aware and Lightweight Middleware for Smart Vehicle Space. In proceeding of the 1st International Conference on Embedded Software and System, LNCS 3605 pp. 266-273 (2004)
3. http://www.omg.org/technology/documents/formal/components.htm. (2005)
4. http://www.cs.wustl.edu/ schmidt/ACE.html. (2005)
5. http://www.cs.wustl.edu/ schmidt/TAO.html. (2005)
6. Qing Wu and Zhaohui Wu: Semantic and Virtual Agents in Adaptive Middleware Architecture for Smart Vehicle Space. In proceeding of the 4th International Central and Eastern European Conference on Multi-Agent Systems, LNAI 3690, pp. 543-546 (2005)
7. Shige Wang, Jeffrey R. Merrick and Kang G. Shin: Component Allocation with Multiple Resource Constraints for Large Embedded Real-time Software Design. In proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium, (2004)
8. Nalini Moti Belaramani, Cho-Li Wang and Francis C.M. Lau: Dynamic Component Composition for Functionality Adaptation in Pervasive Environments. In proceedings of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, (2003)
9. Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten and Betty H.C. Cheng: Comosing Adaptive Software. IEEE Computer Society, pp. 56-64, (2004)
10. Kurt Wallnau, Judith Stafford, Scott Hissam and Mark Klein: On the Relationship of Software Architecture to Software Component Technology. In proceedings of the 6th International Workshop on Component-Oriented Programming, (2001)
11. Qing Wu, Zhaohui Wu, Bin Wu, and Zhou Jiang: Semantic and Adaptive Middleware for Data management in Smart Vehicle Space. In proceeding of the 5th Advances in Web-Age Information Management, pp. 107-116 (2004)