An Adaptive Requirement Framework for SCUDWare Middleware in Ubiquitous Computing

Qing Wu, Danzhen Wang
Institute of Computer Application Technology
Hangzhou Dianzi University
Hangzhou, Zhejiang
P.R. China
http://www.hdu.edu.cn

Abstract: - Due to high dynamic computing environments of ubiquitous computing, it poses many challenges for software middleware technologies. The component-based middleware systems should possess self-adjusting functions for adapt to internal and external environments variation. This paper firstly describes a middleware called SCUDWare for smart vehicle space in ubiquitous Computing. Then we propose a middleware requirement model including users' and resources' variable requirements. After that, a component dynamic behavior model is presented. Next, an adaptive requirement framework is given in detail, which can automatically tune middleware configuration parameters, and conduct a safe and dynamic component composition to preserve the middleware QoS requirements. Finally, it is prototyped and validated by using a mobile music program to analyze performance of this framework.

Key-Words: - **Ubiquitous Computing, Adaptive middleware**, **Component-based technology**

1 Introduction

Ubiquitous computing [1] aims at pursuing naturalness and harmony, which poses software middleware systems operating under highly unpredictable and changeable conditions. Today, a large number of smart and embedded devices come into our life such as mobile phones, PDAs, and smart cameras. Increasingly, the physical world and information space fuse naturally and seamlessly. This computation becomes more embedded and ubiquitous, and provides more facilities and comforts for people. Therefore, it results in many problems in software middleware design, development and running, which should be aware of dynamic computing contexts and could reconfigure its resources to maintain the best application performance in variable environments. We consider 'adaptation' is the key issue for software middleware to meet the changing environments and the diverse run-time contexts. In addition, component-based software architecture provides a novel infrastructure and a development platform for ubiquitous computing. We think that component-based adaptation is more important for software middleware.

On the other hand, vehicles have played an important role in our daily life. People want to require more safety, comfort, and facilities in vehicles. Therefore, we select a vehicle space [2] as a representative scene to study ubiquitous computing. Philip K. Mckinley [3] presents that adaptation should be safe and performed in a disciplined manner. Based on their contributions, our current work focuses at integrating safe and dynamic adaptation into SCUDWare [4] middleware for smart vehicle space. This paper mainly presents an adaptive requirement framework of SCUDWare middleware. Especially, a safe and dynamic component composition method is detailed by analyzing the component run-time behaviors and interdependence relationships. An experiment prototype called 'mobile music system' is built in smart vehicle space to demonstrate the feasibility and reliability of our methods and techniques.

The remainder of the paper is organized as follows. Section 2 presents the SCUDWare middleware platform including smart vehicle space, CORBA Component Model specification overview, and SCUDWare middleware architecture. Then a middleware requirement model, including user and resource requirements, is proposed. In section 4, we present component dynamic behaviors in detail. Then a safe and dynamic component composition method is proposed in section 6. After that, we introduce an adaptive requirement framework of SCUDWare middleware. In section 7, we give a case study and evaluate performance of the framework and methods. Finally, we draw a conclusion in section 8.

2 SCUDWare Middleware Platform

Conformed to the CORBA component model specification, we have built a SCUDWare middleware platform for smart vehicle space naturally and adaptively. We use the adaptive communication environment and it's ORB, which is a real-time object request broker, developed by Washington University. According to the application domain of smart vehicle space, we reduce this ORB selectively and add some adaptive services such as adaptive resource management service, context service, and notification service.

2.1 Smart Vehicle Space

Now many developers have applied embedded, biology authentication and AI technologies to vehicles. The drive capability, dependability, comfort, and convenience of the vehicle are improved greatly. When people go into smart vehicle space, they find many intelligent devices and equipments around them. They communicate with these tools naturally and friendly. It forms a harmonious vehicle space where people, devices, and environments co-operate with each other adaptively.

Smart vehicle space consists of four parts and is defined as *SVS*=(*CA*, *CR*, *AC*, *CP*). *CA* is a context acquisition system. *CA*=(Δ*State*(*pe*,*de*,*en*), (*sen*, *cam*, *sou*)) aims at sensing status changes of people, devices, and environments in the vehicle, including sensors, cameras, and sound receivers. *CR* is a context repository reasoning system. *CR*=(*context*, *ontology*, *domain*, *inference*) uses the correlative contexts and application domain ontology to make the manipulating strategy for adaptation. *AC* is an auto controlling system. *AC*=(*ste*, *com*, *ent*, *nav*, *sec*) consists of steering, communication, entertainment, navigation, and security subsystem. *CP* is a centralized processing system. Particularly, *CP* is a kernel of the smart vehicle space, which controls above third parts co-operating effectively.

2.2 CCM Overview

CORBA is one of software middleware, which provides language and operating system independences. CORBA component model (CCM) is an extension to CORBA distributed object model.

CCM prescribes the specifications of component designing, programming, packaging, deploying and executing stages. CCM specification defines component attributes and ports. Attributes are properties employed to configure component behavior. Specially stated, component ports are very important, which are connecting points between components. There are four kinds of ports: facets, receptacles, event sources, and event sinks. Facets are distinct named interfaces provided by component for client interaction. Receptacles are connection points that describe the component's ability to use a reference supplied by others. Event sources are connection points that emit events of a specified type to one or more interested event consumers, or to an event channel. Event sinks are connection points into which events of a specified type may be pushed.

2.3 SCUDWare Middleware Architecture

SCUDWare architecture consists of five parts defined as *SCUDW* = (*SOSEK*, *ACE*, *ETAO*, *SCUDCCM*, *SVA*). *SOSEK* [5] denotes SMART OSEK, an operating system of vehicle conformed to OSEK specification developed by us. *ACE* denotes the adaptive communication environment, providing high-performance and real-time communications. *ACE* uses inter-process communication, event demultiplexing, explicit dynamic linking, and concurrency. In addition, *ACE* automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads. *ETAO* extends *ACE ORB* and is designed using the best software practices and patterns on *ACE* in order to automate the delivery of high-performance and real-time QoS to distributed applications. *ETAO* includes a set of services such as the persistence service and transaction service. In addition, we have developed an adaptive resource management service, a context service and a notification service. Specially, the context service [6] is based on semantic information. *SCUDCCM* is conformed to CCM specification and consists of adaptive component package, assembly, deployment, and allocation at design-time. Besides, it comprises component migration, replacement, updating, and variation at run-time. In addition, the top layer is *SVA* that denotes semantic virtual agent [7]. *SVA* aims at dealing with application tasks. Each *sva* presents one service composition comprising a number of meta objects. During the co-operations of *SVA*, the SIP (Semantic Interface Protocol) set is used including *sva* discovery, join, lease, and self-updating protocols.

3 Middleware Requirement Model

Middleware requirement model includes user and resource requirement model. QoS is the outcome of the interaction of user, resource requirement, and system behavior. Uncontrollable QoS will be caused by unknowing about any one of three factors. Only accurately describe these factors and their dependency relationship, can the best application performance be achieved.

3.1 User Requirement

User constraints can be divided into soft constraint and hard constraint [8]. Hard constraint emphasizes certain performance index must be achieved. It is the bottom line that user can tolerate. The soft constraint stands for the anticipated application performance. The more the application is close to it, the greater utility this application will achieve, and user will get more contentment from it. The service supplied based on context, mainly focuses on two universal performance indexes:

Definition 1. *The average service-response time* R^s. The average time spent on requesting a service from server and responding to client after processing service.

Definition 2. *The average service throughput* T^s. The amount of the service having been performed in the unit time.

Among all performance indexes, what the users are concerned about is the maximum average response time $R^s_{\max}$ and the minimum average throughput $T^s_{\min}$. $R^s_{\max}$ denotes the maximum average service-response delay time which user can tolerate. $T^s_{\min}$ denotes the lowest capability of service execution which users can tolerate. Thus both $R^s_{\max}$ and $T^s_{\min}$ describe the hard constraint of the application performance. It is described by formula as follows.

$$(R^s \le R^s_{\max}) \cap (T^s \ge T^s_{\min}) \quad (1)$$

It defines performance deviation which expresses the deviation extent of the actual value of performance index to the value of hard constraint. Thus response time deviation here can be defined as follows.

$$\Delta R^s = (R^s_{\max} - R^s) / R^s_{\max} \quad (2)$$

Simultaneously throughput deviation here is defined as follows.

$$\Delta T^s = (T^s - T^s_{\min}) / T^s_{\min} \quad (3)$$

On the basis of performance deviation, it defines function *Utility* to describe the soft constrain. *Utility* here is defined as follows.

$$Utility = \sum_{s \in S} w_s \times (w^R_s \times \Delta R_s + w^T_s \times \Delta T_s) \quad (4)$$

From *Utility* function, which is described by simply using the linear relationship of performance deviation, it can be found that when the response time become less, the throughput will become more, the performance deviation will become bigger and the value of *Utility* will be greater. This analysis shows that the value of *Utility* can directly reflect the satisfaction degree of the soft constraint. In formula *Utility*, *S* denotes the services set which component-based applications provide, and w_s denotes the weight of different service.

$$\sum_{s \in S} w_s = 1 \quad (5)$$

For each service *s*, w^R_s and w^T_s respectively reflect different preference on throughput and response time.

$$w^R_s + w^T_s = 1 \quad (6)$$

3.2 Resource Requirement

For the component technology based on container architecture, each component runs in corresponding container. Application server use container to manage the execution of components. Container processes all component behaviors including interaction with external system and manages all kinds of system resources. Therefore the performance of the component-based application not only depends on itself, but also depends on the middle system where the application is deployed. Meanwhile the performance of middleware system depends a great extent on correct resource parameters configuration. The performance of component behavior depends on resource configuration which its container has applied for it. Concurrently access resource can lead to their competition, and this will have influence on requesting processing. As a result, the most important part of system performance is modeling their relationships.

Definition 3. *Assume that a composite service is composted by n service components, then DG=(SC, E) denotes the function relationship of composite* *service.*

$SC = \{sc_i, 1 \le i \le n\}$, $E = \{(sc_i, sc_j), 1 \le i \le n\}$

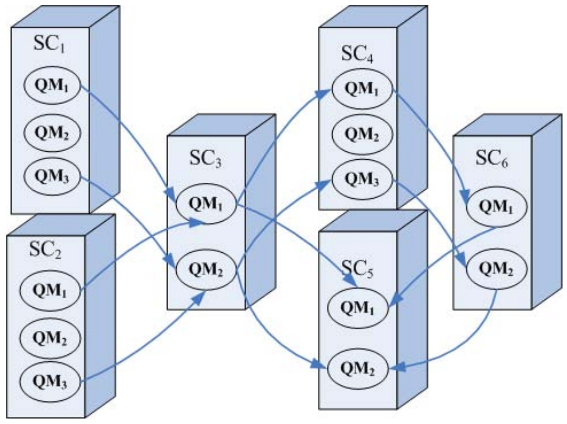

Fig.1 One example of the resource dependency of components.

One example is shown in fig. 1. Every directed edge between two nodes represents existing control-flow or data flow. In addition to meet function dependency between components, each composite service still has to meet QoS requirement between each component. In general, QoS characteristic of component can be described by triple as $QoS = (QoS_{in}, QoS_{out}, \mathrm{Re})$. $QoS_{in} = (q_{in}^1, q_{in}^2, ..., q_{in}^s)$ denotes the QoS list whose adjacent component instance have to be achieved when running a component instance. $QoS_{out} = (q_{out}^1, q_{out}^2, ..., q_{out}^m)$ denotes the QoS list whose adjacent component instance provides when running a component instance. *Re* denotes the demand of system resource like CPU, memory, network bandwidth and so on, when running a component instance. The relationship among QoS_{in}, QoS_{out}, and *Re* is that the attribute value of QoS in QoS_{out} depends on attribute value of correlated *Re* and QoS_{in}. In addition, QM_i denotes the QoS model of component. The different values of QM_i mean the different demands of resource of this component. For example, the CPU utilization of one component can be *10%*, *15%* or *20%,* and then this component has three different QoS models. Under which model the component will run depends on the status of system resource at that time.

Definition 4. $\mathrm{Re} = (r_1, r_2, ... r_k)$ *is a resource requirement list of a service component. Among them, k denotes k different styles of available resource like CPU, memory and network bandwidth and so on.* $r_i \cdot (1 \le i \le k)$, *a specific value, describe the quantity needed for resource of number i like* $r_{cpu}=20\%$ *or* $r_{memory}=2kb$.

Definition 5. *Assume that n service components are running in system and each component has k different styles of resource requirement, then the total of resource of these n components request can be described in following formula.*

$$\sum \mathrm{Re} = (\sum_{i=1}^{n} r_1^i, \sum_{i=1}^{n} r_2^i, ..., \sum_{i=1}^{n} r_k^i) \tag{7}$$

Definition 6. *Assume that n service components are running in the system, available resources can be described like RA=(ra₁, ra₂,…, ra_k,). And the needed resource of service component* $\sum \mathrm{Re}$ *is calculated according to formula (7). If these n service components can be instantiated and can run in the system, following formulas holds.*

$$(\sum \mathrm{Re} \le RA) \cap (\forall_{j\cdot(1 \le j \le k)} \cdot (\sum_{i=1}^{n} r_1^i \le ra_j)\cdot) \tag{8}$$

In other words, only resource requirement of running service component in equipment be satisfied as above inequality, this service could be deployed successfully.

Definition 7. *A service composted by n service components has k different styles of resource requirements, and then SRC is defined as follows.*

$$SRC = \sum_{i=1}^{k} w^i \times r^i + \sum_{1 \le i, j \le n}^{i \ne j} w^k \times r_{sc_{ij}}^k \tag{9}$$

Among them, w^i denotes the resource weight which means importance degree of *i* resource. This indicates that the value of w^i become bigger, this kind of resource will have more difficulty in applying. $\sum_{i=1}^{k} w^i = 1 \cdot (0 \le w^i \le 1)$. sc_{ij} denotes dependency relationship between component sc_i and sc_j. If sc_i and sc_j are deployed on same equipment then $r_{sc_{ij}}^k = 0$, otherwise $r_{sc_{ij}}^k$ denotes the network bandwidth occupied by these two components for their communication. The value of *SRC* depends on r^i, weight *w* and $r_{sc_{ij}}^k$. In general, the higher demand level of QoS, the better quality of the service, the more demand quantity for resource, the bigger value of SRC. But the distributed resource is often very limited and therefore the amount of distributed resource is often less than the best quality of service demands. From the above analysis, it can easily be found out that when the value of SRC becomes bigger, the time waiting for satisfying with needed amount of resource will be longer. Waiting time become longer means the response time of service increases and the quality of service decreases. As a result, the value of SRC can be used as a reference point for assessing resource allocation strategy.

4 Component Dynamic Behavior

In terms of the resource abstract framework [9], we develop an adaptive component management service for the safe and dynamic component composition. According to the changing run-time contexts, this service is responsible for managing the component behaviors in an appropriate way including component addition, removal, updating, replacement, and transfer. This section discusses the adaptive component model and its run-time behaviors.

4.1 Adaptive Component Model

Adaptive component, the key element of middleware, is encapsulated as an entity communicate with outside only with interface. It divides two species of single component and composite component.

Definition 8. *Adaptive component t::= single component | composite component*

Port consisted by channels constitutes interface. Channel plays an important role to receive and sent data information. Connector is deemed as a special component takes a role as bridge between components. The dynamic requirement is concerned with the following systems structural changes: change of the connection between elements in the system, create and delete new components as well as new ports and channels dynamically. Garamarical specification of dynamic behavior of component is described as follows:

Definition 9. *choreographer* ::= *attach.choreographer* | *detach.choreographer* | *Create.choreographer* | *Destroy.choreographer* | *Ecomputation.choreographer* | ***inaction*** | ***replicate***

Definition 10. *detach* ::= ***detach*** *ComponentInstanceName^ PortName* ***from*** *ConnectorInstanceName ^ PortName* | ***detach*** *ComponentInstanceName ^PortName ^ ChannelName* ***from*** *ConnectorInstanceName ^ EportName ^ EchannelName*

Attach is used to create new connector, detach to delete connector. They are both modeled as α change-name operation in high-order multi-type π calculus.

Definition 11. *Create* ::= ***new*** [*01 ComponentInstanceName:*] *ComponentName*([*0 + Actual-parameter*]) | ***new*** [*01ConnectorInstanceName*:] *ConnectorName* ([*0 + Actual-parameter*]) | ***new***[*01 PortName*:] *PortType name* | ***new*** [*01PortName ^ ChannelName*:] *ChannnelType name*

Component behavior is modeled as process P_a. $P_a = P_{a1} + P_{a2}$. P_{a1} represents component's computing behavior (routing behavior), P_{a2} means evolving behavior predefined by choreographer specification. Running-environments behavior is modeled as P_b in high-order multi-type π calculus. ζ represents the virtual channel for communication between component and running environments. If choreographer specification intends to create new component instance or connector instance, following process formula, according to high-order π calculus, expresses it well.

$$P_{a2} = \zeta(x)\cdot(x \mid P_{a1} + P_{a2}), P_b = \bar{\zeta}(P_c)\cdot P_b$$

P_c represents behavior of pending component preparing to create. If choreographer specification intends to create new port or channel.

System behavior means composite component behavior combines running-environments behavior to run concurrently. With choreographer to create component instance or connector instance, system behavior can be described like this:

$$P_a \mid P_b = (P_{a1} + P_{a2}) \mid \bar{\zeta}(P_c)\cdot P_b$$

$$P_a \mid P_b = (P_{a1} + \zeta(x)\cdot(x \mid p_{a1} + P_{a2})) \mid \bar{\zeta}(P_c)\cdot P_b$$

According to specification *R-COM*, a specification from high-order multi-type π, above-mentioned formula can convert to this one:

$$(P_{a1} + P_{a2})\{0/x\} + P_b$$

Composite component (composite connector) behavior evolves to this one: $(P_a|P_{a1}+P_{a2})$, meanwhile computing behavior of composite-component can evolve from P_{a1} to $P_c|P_{a1}$. This evolution means composite-component (composite-connector) creates a new component instance (new composite-connector instance) whose behavior is P_c. With Choreographer to create port and channel, system behavior can be described like this:

$$P_a \mid P_b = (P_{a1} + P_{a2}) \mid \bar{\zeta}(0)\cdot P_b$$

$$P_a \mid P_b = (P_{a1} + \zeta(x)\cdot(x \mid p_{a1} + P_{a2})) \mid \bar{\zeta}(0)\cdot P_b$$

Above formula also can evolve to this one:

$$(P_{a1} + P_{a2})\{0/x\} \mid P_b$$

It means composite-component (connector) create a new port (channel) 0. Destroy is used to delete component instance, connector instance, port and channel. With following the thought of dealing with Create, Destroy also can be explained as process in high-order π calculus.

4.2 Component Run-time Behaviors

According to the CCM specification, there are five essential component run-time behaviors in SCUDWare. Those are component addition, removal, updating, replacement, and transfer, defined as *CoB*={*CoA, CoRv, CoU, CoRp, CoT*}. In SCUDWare, components fit together to be a component composition. For adaptively executing tasks, components in this composition perform the relative behavior actions. As following, we give some formal definitions of the component run-time behaviors.

Definition 12. *A Component Composition Unit is defined as CCU*=(*AC, CRIG, CONTEXT*). *AC* denotes a set of adaptive components composition. *CRIG* is the component interdependence graph introduced in the section 4.3. *CONTEXT* denotes the run-time environments of application tasks.

Definition 13. *A Component Composition Group is defined as CCG*={(*CCU, T*)}.

$$\exists t \in T \cdot \forall g \in CCG \to \exists c_i, c_j \in CCU \cdot (c_i \wedge c_j = \phi)$$

At time *t*, each component composition group comprises one or more component composition unit. Any two component composition unit has not the same component. That also means a component is exclusively in one component composition unit at one time.

Definition 14. *CoA* (*Component Addition*) *is defined as CoA*(*AC,CCU*). After a component is implemented according to the CCM specification, and tries to join into one CCU of the SCUDWare, the component addition behavior will be performed. Specially stated, there are two important principles during the execution. (1) The addition behavior must not break current other components execution. (2) The component interdependence among this new component and other old components of this CCU should be decided immediately.

Definition 15. *CoRv* (*Component Removal*) *is defined as CoRv*(*AC, CCU*). If a component in one CCU is not needed any more, this component can be removed from this CCU. Importantly stated, before performing this behavior, ACMS should check that the whole components in this CCU do not depend on this component any more. As long as one component still needs this component, that component can not be removed.

Definition 16. *CoRp* (*Component Replacement*) *is defined as CoRp*(*AC, AC', CCU*). If a component in one CCU can not satisfy the changing run-time contexts or application requirements, another appropriate component will replace this component for adaptation.

Definition 17. *CoU* (*Component Updating*) *is defined as CoU*(*AC, AC', CCU*). Specially, AC' is the new version of AC. We can classify this behavior to the CoRp. However, CoU realizes component self-updating.

Definition 18. *CoT* (*Component Transfer*) *is defined as CoT*(*AC, CCU, CCU'*). If a component in one CCU is not needed, but another CCU' demands it, component transfer behavior will be executed. Importantly, the component interdependence this component in CCU should be removed. Also, the component interdependence of this component in CCU' should be decided simultaneously.

4.3 Component Run-time Interdependence Graph

In SCUDWare, *CCU* is responsible for one special task. All components in the *CCU* are cooperating and communicating with each other. Therefore, there are interdependences among those components in the *CCU*. In order to define the inter dependence relationships among components; we introduce a component run-time interdependence graph (*CRIG*) composed of component nodes and link paths.

For components in one *ccu*, we first associate a node for each component. Two component nodes are linked, and the link path is labeled with a weight. Formally, we define a component interdependence graph as *CRIG* = (*CCU, LP, W*).

(1) *CCU* includes a set of component nodes.

(2) $LP = \{l_{ij} (1 \le i \le n, 1 \le j \le m)\}$ denotes a set of component links, describing the dependent targets. l_{ij} is a link between the component nodes: cn_i and cn_j.

(3) $W = \{w_{ij} (1 \le i \le n, 1 \le j \le m)\}$ denotes a set of interdependent weights. w_{ij} is a non-negative real number, which labels l_{ij}.

Importantly, w_{ij} reflects the importance of the interdependence between two associated components. These weights used, for example, to detect which links becomes too heavy, or whether the *ACMS* relies too much on some components. In terms of this weight, we can decide which component can be removed as long as no component needs it any more. Extremely, *CRIG* changes according to the different contexts and application requirements. Therefore, these interdependencies are not static.

It can be modified when a new component is added, or one component is removed, or component transfer is performed.

5 Safe and Dynamic Component Composition

In this section, we present a safe and dynamic adaptation computation method. We first give the definitions of the *CCU* and *CCG* states. Second, we introduce a dynamic component composition state transition net. Finally, the process of safe and dynamic component composition is presented.

5.1 CCU and CCG States

In order to dynamically compose the components safely, we give the following definitions of the *CCU* and *CCG* states, consisting of *CCU* Non-Risk State, *CCU* Underlying Risk State, *CCU* Unsafe State, and *CCG* Safe State.

(1) *A CCU Non-Risk State* is defined as *NRS*(*ccu*)=*True* if $\exists ccu, crig \forall w \in crig \cdot (w < \alpha)$.

In one *ccu*, if any *w* is less than *α*, this *ccu* is at a *Non-Risk State*. *α* is a pre-defined constant describing the maximum component interdependence limit.

(2) *A CCU Underlying Risk State* is defined as *URS*(*ccu*)=*True* if $\exists ccu, crig \exists w \in crig \cdot (w \geq \alpha)$.

In one *ccu*, if at least one *w* is more than *α*, this *ccu* is at an underlying risk state. In this state, there are possibilities that some unpredictable errors occur in the systems.

(3) *A CCU Unsafe State* is defined as *CUS*(*ccu*)=*True* if $\exists c_i \in ccu \wedge l_{ij} \in crig \wedge (USE(c_j) = false)$

This state means that component c_i of one *ccu* depends on component c_j, but component c_j is not usable. We use $USE(c_j)=false$ to define that c_j is not usable. Commonly, there are two conditions may result in this state, which are (1) c_j is in this *ccu*, but is error, (2) c_j is removed from this *ccu*.

(4) *A CCG Safe State* is defined as *GSS*(*ccg*)=*True* if

$$\forall ccu \in ccg \cdot (NRS(ccu) = true) \wedge (URS(ccu) = false)$$
$$\wedge (URS(ccu) = false)$$

If one *ccg* at a Safe State, all the *ccu* in this *ccg* should be at a non-risk state, and not an underlying risk state, and not an unsafe state.

5.2 Dynamic Component Composition State Transition Net

Component run-time behaviors may cause the component state transition. During the executions of the component run-time behaviors, two principles should be conformed, which are: (1) It should not break down the existent component interdependencies. (2) The action of component run-time behavior is an atom and integrity operator. We should assure that the component composition enters a safe state. Otherwise, it may cause an Unsafe State and make the system break down.

In our work, Petri net is used to describe the dynamic component composition state transition. We define the state transition net as *DCCSTN* = (*P, T, I, O, M, R*). *P* denotes a set of places of *DCCSTN*, and *P* equals to the states of *CCU*. *T* denotes a set of transitions of *DCCSTN*, and *T* equals to *CoB*.

$I \subseteq P \times T$ is a set of input functions. $O \subseteq T \times P$ is a set of output functions. *M* denotes a set of time consumptions of *T*. *R* denotes a set of computation resource consumptions of *T*.

During the dynamic component composition, in order to assure the safety, we can find a safe and effective composition solution according to *DCCSTN*. On one hand, in terms of *DCCSTN*, we can know which executions of component behavior will result in a unsafe state. These executions should be avoided by *ACMS*. On the other hand, according to *DCCSTN*, considering the time and computation resource consumptions of each execution, *ACMS* can select an effective composition solution. For example, *ACMS* can find an appropriate solution of component composition, which has the minimum communication resource consumption, and the minimum component behavior execution time.

5.3 SDCC: Safe and Dynamic Component Composition Process

In SCUDWare, *ACMS* is responsible for the whole *SDCC* process. Once *ACMC* monitors one *ccu* that satisfies *URS*(*ccu*)=*true* and enters an underlying risk state, it will perform appropriate component run-time behaviors to eliminate this underlying risk. There are four steps of *SDCC* process.

(1) *Preparing Step*. Before the execution of *SDCC*, *ACMS* should check the run-time interdependence relationships of the components those are the object of the execution.

For example, before *CoRe*(c_1, *ccu*) is performed, *ACMS* firstly decides the run-time interdependence of c_1. Assume that component c_2 and c_3 both depend on c_1, *ACMS* will send a *c_1 removal request message* to c_2 and c_3. Similarly, there are other types of request message such as component addition, updating, replacement, and transfer.

(2) *Waiting Step*. After sending the request message to the object components that acquire dependence, *ACMS* will wait for the reply messages. Commonly, there are two conditions for the object component. (a) This object component is running. (b)This object component is not running. As for (a), *ACMS* can not receive the reply message until the object component completes execution. As for (b), *ACMS* can get the reply at once. In addition, due to the unknown faults, we introduce a timeout and re-send mechanism. That means if *ACMS* has not received the reply message for a limited time, *ACMS* will re-send the request message to this object component. If the number of re-sending is more than a pre-defined number, *ACMS* will consider that the object component is not running and it can continue and go to next step.

(3) *Performing Step*. Once *ACMS* receives all the replies from the object components, it begins to perform the actions of the component behaviors. *ACMS* will find a safe and effective composition solution according to *DCCSTN*. The particular method is presented in section 6.3.

(4) *Updating Step*. This is the last step of the *SDCC* process. After *ACMS* finishes the actions of component behaviors, it will update the inter-dependence relationships among the components in this *ccu*. Importantly stated, the component interdependence in the old *ccu* and the new *ccu* should be updated simultaneously.

6 Adaptive Requirement Framework

In this section, we will introduce the architecture of adaptive middleware framework, and give a middleware parameters configuration algorithm and a safe and effective dynamic component composition method

6.1 Architecture of Adaptive Requirement Framework

Figure 2 shows an adaptive requirement framework of SCUDWare middleware, which consists of load monitor, configuration selector, component composition module, and a QoS monitor.

QoS monitor is responsible for calculating the amount of completed service requests, each service's response time, the average service-response time and throughput in every adjusted time interval. In addition to that, it also checks whether the QoS requirement is violated or not. All these statistic data will together determine when the application server needs re-configuration. When application need re-configuration, QoS monitor will inform configuration selector and component composition module. After received re-configuration order, configuration selector will search configuration parameters in parameter table and select the most satisfactory configuration after considering each candidate configuration and dependency relationship between these parameters. Component composition module will conduct dynamic component re-composition actions. Load monitor is in charge of calculating the amount of each service requests and partly control configuration selector according to user demand.

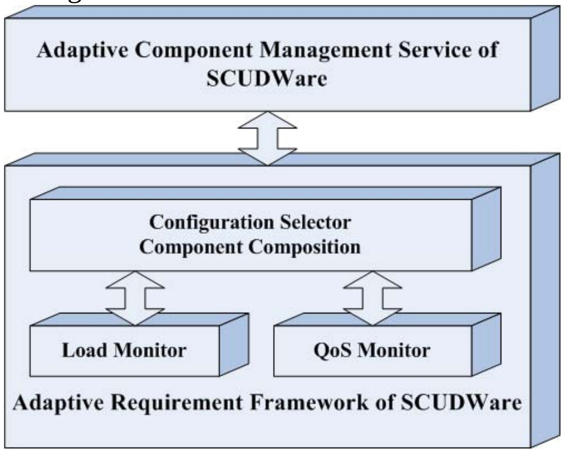
Adaptive Component Management Service of SCUDWare
Configuration Selector
Component Composition
Load Monitor
QoS Monitor
Adaptive Requirement Framework of SCUDWare

Fig.2 Adaptive requirement framework of SCUDWare middleware.

6.2 Middleware Parameters Configuration

The goal of adaptive middleware framework is that under changeable load, when the hard constraint and resource constraint of each service has been satisfied, making soft constraint achieve the highest utility value through adaptively adjusts the related server configuration parameters. About this adjusting, the key is how to determine the correct configuration parameters. The framework uses following algorithm to search a point in configurable parameters table of server in order to meet hard constraint and maximize the utility of QoS. *K* dimensional vector *Re*=(r_1,r_2,…r_k) denotes a kind of configuration defined Re contains *k* parameters and each parameter value has a given range. $r_i \in (QM_i^1, QM_i^2, ..., QM_i^n)$ means that each component demand for QoS model is not necessarily same. Thus here the value of n is also not necessarily same. Adjacent configuration set of Re can be described as follows.

$$M_{\mathrm{Re}} = \{\mathrm{Re} + t_i\} \cup \{\mathrm{Re} - t_i\} \qquad (10)$$

t_i is *k* dimensional vector equals {*0,…1,…0*}. The value of *i* element is *1*. And the remaining is *0*. Re_0, a configuration parameter before adjustment, is the initial focal point in searching. In iteration, this algorithm will select the best configuration among adjacent configuration to focal point and obtain component interaction relationship which mainly contains dependency relationship including call sequence, call amount and degree of concurrency of call.

This algorithm will check resource constraint. If resource constraint is satisfied, then the component will be instanced, else the algorithm continues to search another configuration. In the search process, the algorithm calculates the average service response time R^s and the average service throughput T^s in each candidate configuration. Vector $R = (R_{s1}, R_{s2}, ...R_{sn})$ is the average service-response time of s_1, s_2,…, s_n. Vector $T = (T_{s1}, T_{s2}, ...T_{sn})$ is the average service throughput of s_1, s_2,…, s_n. The utility value based on this configuration, is calculated by function Utility. HC denotes the bottom line of performance user can tolerate. Function *satisfy_user* is used for judging whether $< R^s, T^s >$ meets *HC* constraint. *HU* denotes needed resource used for completing component deployment. Function *satisfy_resource* is used for judging whether $\sum \mathrm{Re} \le RA$ meets resource constraint. RA=(ra_1,ra_2,…,ra_k). Function *Predict* is used for calculating service performance. The algorithm is described as follows.

```
START;
Dres=1;
$Re_{cur}=Re_0$, $Re_{new}= Re_0$;
REPEAT
Improved=FALSE;
$< R^s, T^s >$=Predict($Re_{cur}$);
MaxUtility=Utility$< R^s, T^s >$;
FOR Each Re in $M_{Renew}$
    $< R^s, T^s >$=Predict(Re);
    IF ( satisfy_user($< R^s, T^s >$, HC)
          AND satisfy_resource(Re, HU)
          AND Utility($< R^s, T^s >$)>MaxUtility )
    THEN
        MaxUtility=Utility($< R^s, T^s >$);
        $Re_{new}$ = Re;
        Improved= TRUE;
    END IF
  END FOR
$Re_{cur}$ = $Re_{new}$;
Dres = Dres + 1;
UNTIL ( (Improved=FALSE)  OR
          (Dres=MaxDres) );
END
```

It's easy to realize using this algorithm, and the convergence speed is almost same as other intelligent search methods, when the state space is relatively small. The operation efficiency is affected by amount of the equipment and component. After configuration, which is evaluated by *SRC* value, the paper sets up the weight according to the actual resource.

6.3 SEDCC : Safe and Effective Dynamic Component Composition Method

According to section 5 analysis, this section presents a safe and effective dynamic component composition algorithm, called *SEDCC* and shown as following.

(1) According to the whole *ccu* and the actions of *CoB*, we build the *DCCSTN*=(*P, T, I, O, M, R*)

(2) Generating the sub net that satisfy *GSS*(*ccu*)=*TRUE* in *DCCSTN*.

(3) *CoB* in *sub-DCCSTN* are generated by *ACMS*. All the costs of *CoB* are calculated by *ACMS*.

(4) Select one appropriate component behaviors in *CoB* based on three principles: (a) The cost of the action execution time should be minimum. (b) The computation resource constrains should be satisfied. (c) The component interdependence should not be broken down.

(5) A set of actions of component behaviors satisfying above three principles is build to form a safe and effective dynamic component composition solution. Then *ACMS* will execute these actions in turn.

7 Case Study and Performance Evaluation

To verify the *SDCC* method including the *SEDCC* algorithm, we have made some preliminary experiments using *ACMS* to build a mobile music system *MMS* in the smart vehicle space.

In *MMS*, components are responsible for acquiring, playing, transmitting, and outputting the music information. These components interact with the request and reply process. If one component sends the request for some music information, *ACMS* will select one appropriate component to work and reply to the demander. Our experiments are tested on the following platforms, as shown in table 1.

Table 1 Experiments test bed.

	HP iPAQ Pocket PC H5500	PC
CPU	Intel 400 MHz, XScal-PXA255	Intel PIV 2.4G
Memory	128 MB RAM + 48 MB Flash ROM	256 MB RAM
Network	Wireless LAN 802.11b	LAN 100MB/s
OS	Familiar Linux v0.8.0-rc1	RedHat Linux 9.0 (2.4.20)
Middleware	SCUDWare	SCUDWare

The iPAQ is connected to the PC via the wireless LAN using 802.11b protocol. The middleware platform is the SCUDWare. The experiment in *MMS* runs on two PDAs and one PC. On the PC, a music producer is placed. It can transfer the music by stereo tune or mono tune. Two PDAs have limited computation and communicated resources, those are connected with PC via the wireless LAN 802.11b. They can receive the music information from the PC, and play music in a proper model such as stereo tune or mono tune. We have developed eight components for this experiment.

The PC can have two components: A_1, a music sender of stereo tune sender and A_2, a music sender of mono tune. PDA_1 can have three components: B_1, a music player of stereo tune, B_2, a music player of mono tune, and B_3, the other music player of stereo tune. PDA_2 can have three components: C_1, a music player of stereo tune, C_2, a music player of mono tune, and C_3, the other music player of stereo tune. In the beginning, we allocate A_1 into the PC, B_1 into the PDA_1, and C_1 into the PDA_2. At first, *MMS* runs well. After some time, the network bandwidth decrease. It is too low to input and output by stereo tune. For adapting to this status change, *ACMS* will perform dynamic component composition. The goal of the experiment is using *SDCC* method to find a safe and effective solution consisting of a component behavior sequence. *ASMS* firstly build the *DCCSTN*. Next *ASMS* calculates and finds an appropriate solution according to above three principles. Finally, these component behavior actions are execute by *ASMS* in turn.

Table 2 CoB time costs, resources consumptions, and effects on component interdependences.

No.	CoB	Time Costs	RC	ECI
(1)	CoRv(A_1,CCU_{PC})	10	0	breaking
(2)	CoA(A_2, CCU_{PC})	10	10	no effect
(3)	CoRv(B_1,CCU_{PDA1})	5	0	no effect
(4)	CoA(B_2,CCU_{PDA1})	10	10	no effect
(5)	CoA(B_3,CCU_{PDA1})	10	12	no effect
(6)	CoRv(C_1,CCU_{PDA2})	5	0	no effect
(7)	CoA(C_2,CCU_{PDA2})	8	10	no effect
(8)	CoA(C_3,CCU_{PDA2})	10	10	no effect

Table 2 shows the each time cost of *CoA*, the sum of resources consumptions *RC*, and the effects on component interdependences *ECI*.

According to the *SEDCC* algorithm, *ACMS* eventually get the safe and effective dynamic component composition solution that is a *CoB* sequence: (2) (3) (4) (6) (7).

In order to test the performance of the *SDCC* method using by *ACMS*, we have the experiments performance evaluations. As shown in fig. 3, we compare with two kinds of the successful execution ratio of *MMS*. One kind uses the *SDCC* method, and the other does not use this method. Specially, we consider three conditions: (a) decreasing network bandwidths, (b) decreasing memory size, and (c) decreasing CPU computation. In addition, we vary the component number from 10 to 100, and the step is 10. In this experiment, we use a random method to make above three conditions. From the fig. 3, we can conclude that successful execution ratio with *SDCC* method is more than that without *SDCC* method under the changing conditions. Therefore, the experiment results show that our method is flexible and adaptive.

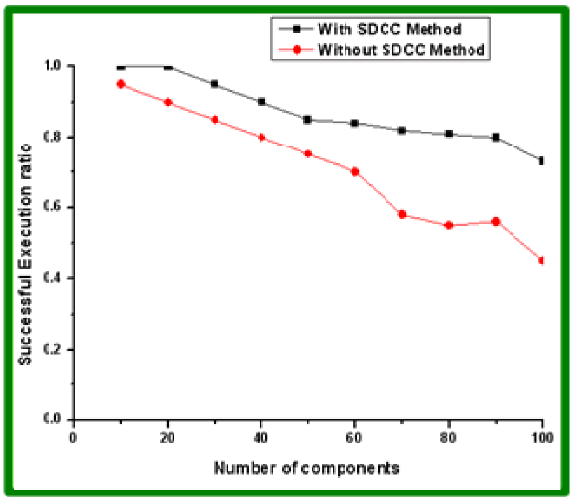

Fig.3 Comparing (a) successful execution ratiowith *SDCC* method, with (b) successful execution ratio without *SDCC* method under the different conditions.

8 Conclusion

In ubiquitous computing environments, that integrating safe and dynamic adaptation into adaptive middleware is playing a more important role. In this paper, we firstly present the SCUDWare middleware platform including smart vehicle space, CCM specification overview, and SCUDWare middleware architecture. Next, a middleware requirement model, including user and resource requirements, is proposed. And then, we mainly introduce an adaptive requirement framework of SCUDWare middleware. In addition, we give a case study and evaluate performance of the framework and methods.

Our future work is to improve performance of this adaptive requirement framework. In addition, we will take other methods to realize more middleware flexibility and reliability for different requirements.

Acknowledgments

This research was supported by National Natural Science Foundation of China under Grant No. 60703088.

References:

[1] Weiser M, The Computer for the 21st Century, *Scientific American*, 1991, pp. 94-100.

[2] Qing Wu, Zhaohui Wu, Bin Wu, and Zhou Jiang, Semantic and Adaptive Middleware for Data management in Smart Vehicle Space, *In proceedings of the 5th Advances in Web-Age Information Management, LNCS 3129*, 2004, pp. 107-116.

[3] Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley, Adding Safeness to Dynamic Adaptation Techniques, *In proceedings of Workshop on Architecting Dependable Systems*, 2004.

[4] Zhaohui Wu, Qing Wu, Hong Cheng, Gang Pan, and Minde Zhao, SCUDWare: A Semantic and Adaptive Middleware Platform for Smart Vehicle Space, *IEEE Transactions on Intelligent Transportation Systems*, Vol.8, No.1, 2007, pp. 121-132.

[5] Mingde Zhao, Zhaohui Wu, Guoqing Yang, Lei Wang, and Wei Chen, SmartOSEK: A Dependable Platform for Automobile Electronics, *In proceedings of the first International Conference on Embedded Software and System*, 2004, pp. 437-442.

[6] Qing Wu and Zhaohui Wu, Integrating Semantic Context Service into Adaptive Middleware for Ubiquitous Computing, *In "Advances in Computer Science and Engineering Series"*, Imperial CollegePress, 2005, pp. 222-231.

[7] Qing Wu and Zhaohui Wu, Semantic and Virtual Agents in Adaptive Middleware Architecture for Smart Vehicle Space, *In proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems*, *LNAI 3690*, 2005, pp. 543-546.

[8] Nikhil Barthwal, Murray Woodside, Efficient evaluation of alternatives for assembly of services, *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, pp. 275-282.