A Case-based Component Selection Framework for Mobile Context-aware Applications

Fan Dong, Li Zhang, Dexter H. Hu, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
{fdong, lzhang2, hyhu, clwang}@cs.hku.hk

***Abstract*—In the ever-changing pervasive computing paradigm, applications, especially those running on resource-scarce mobile devices, have to adapt to the runtime environment as the users are roaming around. Various adaptation techniques, relying on dynamic composition of components, have been proposed by a number of researchers. Nevertheless, most existing approaches only support component selection based on predefined rules and strategies. Because of the limitation of pure rule-based approach, context-awareness can not be well supported. In this paper, we propose a software component selection framework for mobile pervasive computing. Our approach adopts the case-based reasoning technique to provide proactive component selection. Context-awareness and personalization are embodied in the reasoning and selection process. As a proof of concept, we developed and evaluated a context-aware personal communicator (CAPC) application using *adaptive component selection*, with a synthesized execution trace obtained from real-life E-mail softwares ported to CAPC. Our results show that the adaptive component selection can reduce maximum memory consumption by at least 20%, and the *context-guided reasoning* technique can improve reasoning accuracy by nearly 10% within acceptable reasoning time.**

I. Introduction

Nowadays, more and more mobile devices, taking advantage of the wide coverage of wireless networks, have been connected to the Internet. The whole environment can be seen as a large-scale ad-hoc distributed system with a multitude of small devices moving from place to place. Due to the unpredictable nature of the mobile computing environment, the future programming infrastructure should provide the ability for applications to adapt their functionality according to the changing contexts. The component-based software architectures, such as .NET, Enterprise JavaBeans, and CORBA, are the most dominant engineering paradigms in current software community and industry, as they allow services or applications to be dynamically assembled at runtime through a process known as *late binding*.

However, traditional component-based softwares do not take context information into account. They require users' or programmers' intervention to specify the components. Some renowned pervasive computing middlewares, such as Gaia [1] and PCOM [2], provide generic adaptation support via either static mapping or predefined rules and strategies. Such rule-based approaches are based on triggering mechanism, where certain context inputs will fire a set of associated rules. They generally follow the "stimuli-response" paradigm, where a set of event-condition-action rules are predefined to trigger actions or code segments. For example, an *"On event IF condition THEN action"* pattern is scattered in the application program to change the execution logics in an ad-hoc manner. However, all triggering conditions must be strictly satisfied or exactly matched to activate the adaptation, which is less flexible in the highly dynamic pervasive computing environment. A pervasive computing system, that strives to be minimally intrusive to end users, needs to be context-aware [3]. However, to realize "deep" context awareness, we need to consider a wide spectrum of context types, ranging from the low-level physical context (e.g., available memory, battery power, and CPU speed), to the high-level user context, including user's physiological and phycological status (e.g., body temperature, emotions, behavioral pattern, and social relationships).

Designing a complete rule-based system concerning all types of contextual parameters to realize true context awareness is not feasible. The system designer can not foresee all possible conditions to be encountered and pre-determine all the rules. There could be still many factors that might influence end user's expectations from the execution environment, which can not be modeled at the design phase. Failure in rule condition matching leads to malfunction of the whole system, which in turn distracts user's attentions. Recently case-based reasoning (CBR) technique has gained interests and adoption in the design of context-aware systems ([4], [5]) and autonomic systems [6].

The CBR technique is capable to produce useful results even with partial information that may not match the past case exactly. Such properties make CBR possible to reason about context and ambient situation, even in some ill-defined and poorly structured domains.

In this paper, we propose a distributed component selection framework for mobile devices in the pervasive computing environment. We adopt the case-based reasoning technique to provide proactive component selection. The framework is built atop of our component-based software infrastructure called Sparkle [7]. Applications are built from small compo-

This work is supported by National Natural Science Foundation of China Grant No. 60773089 and Grant No. 60533040.

nents, called *facets*, which can be downloaded, assembled and even discarded on demand. Thus, dynamic adaptation can be achieved via component composition and reconfiguration at runtime. Context-awareness and personalization are embodied in the reasoning and component selection process.

We identified several new challenges while applying the CBR technique to a component selection framework:

1) The component selection framework has to cope with more complex context information (i.e. user context, location context, device context and environment context) than the existing recommendation systems [8]. Therefore we must have a formal way to model the context and reason about the context. It also leads to more complex case structure in terms of the complexity in representing context attributes.
2) The *facet* model enforces a tree structure representation to show the calling dependencies among components. Possible combinations on the selection of components and unpredictable execution order with feedbacks may lead to a large case size, which could hinder the performance of the case-based reasoning process.
3) Compared with other case-based recommendation systems, what we recommend or provide is a software component that can fulfill certain functionality. The responsibility to find and return a component suitable for the device configuration and current context should not rest on the users. As users may not have much knowledge or expertise on the component itself.

The rest of this paper is organized as follows: Section II describes the related work. Section III gives an overview of the system. Section IV - VI explain the details of how to model the cases using OWL, how to realize adaptive component selection. Reasoning accuracy and performance valuations are reported in Section VII. Section VIII gives the conclusion and future work.

II. Related Work

A. Component models and platforms

Many existing works realize self-adaptation through dynamic composition and reconfiguration of components or services. UIUC's Gaia [1] is a programming environment (active space) extending the model-view-controller pattern. It was built based on OMG's CORBA component model. When a user moves from one active space to another, the system uses two external mappings to achieve dynamic adaptation based on non-functional parameters. Gaia adds a level of indirection between I/O device and traditional application. The system assumes a relatively static environment, only simple reasoning mechanism was adopted. Compared with Gaia [1], PCOM [2] provides better granularity by breaking apart atomic application into components connected as a tree, where the links show their calling dependencies. Automatic adaptation is supported by selecting the optimal component at runtime. In case multiple components fulfill the requirement, predefined user preferences are adopted. StarCCM [9] proposed a policy-based context-aware middleware support for component-based applications in pervasive computing. It does not mention about mobile scenarios and therefore the context is rather static. Besides, it does not show what kind of context should be taken into account and how to model the context. D. Ayed, et al. [10] proposed a two-level rule-based adaptation approach for the deployment of mobile component-based applications. Apart from the target deployment environment, it suggests that the user's preferences, activities and physical environment should also be considered. Nevertheless, it does not give any formal guide on how to model the context and link the context model with the adaptation process.

B. CBR in context-aware applications

It is commonly agreed that context-aware applications can dynamically adapt to the environment whenever it changes. Ander K. Petersen et al. [11] suggest that the case-based reasoning supported by a rich knowledge model, is a promising approach to assess situation by being context-aware. In [4], Ma et al. showed how to use CBR as context-awareness solution in smart home. CBR is used to adapt services of smart home to user's preferences. User's preferences are captured in the cases. Whether a proposed solution is correct or not is justified by observing how a user interacts with the system. Their approach however required a lot of sensors embedded in the smart home environment to monitor the user's behavior. AmbieAgents [12] is a scalable infrastructure for mobile and context-aware information services, which employed GREEK [13], a case-based reasoning engine to determine the user situation. A detailed account of the use of CBR in a multi-agent system can be found in Ander K. Petersen et al. [14]. The work discussed how to use a multi-agent system to offer user with personalized and context-sensitive information. Ander K. Petersen also discussed four challenges in the design of CBR for achieving context awareness in ambient intelligent systems in [11], which inspired some of our thoughts. Apart from CBR's intensive application in smart home and information service applications, CBR technique also plays an important role in various recommendation systems [8]. In [15], the authors proposed a novel distributed service discovery and selection framework in a pervasive service environment that consists of service clustered servers (SCS). The working mechanism of this work is very similar to that in the Sparkle infrastructure. The service in SCS is mainly referred to media service, web service and document service. When a service is needed, the user request is forwarded to the local SCS for processing, similar to the proxy in Sparkle. If local SCS can not locate the service, the request is passed on to the domain facilitators who have a better view of the entire domain. The service selection is based on case-based reasoning technique. The system adopts two-level similarity measurements, namely local and global. However, only the similarities of service attributes or service related contexts (i.e. type, location, cost and bandwidth) were computed. More high-level user context information was not taken into account.

III. System Overview

To provide more personalization and context-awareness, we present a distributed component selection framework for mobile context-aware applications based on case-based reasoning technique. Our framework is built atop of the Sparkle software infrastructure [7], where application program can be assembled by components or modules. Sparkle consists of a client system embedded in the mobile device and an intelligent proxy server that can be connected to the Internet. The existence of component providers and context providers is a premise for our framework. We leverage the OWL/RDF from semantic web community to model the context and domain knowledge. We believe that the component-based software model assisted by CBR is a promising way to achieve personalization and context-awareness in the pervasive computing environment. Traditionally, applications are programmed in a monolithic way. These softwares become too big to fit into the small devices with limited resources. Functionalities that could be provided by a device are therefore restricted by its configuration. In Sparkle, applications are built from small components, called *facet*. To fulfill the same functionality, there could be multiple candidate facets. These components are downloaded from the network at run-time, and could be cached for future use or thrown away after use. Thus, applications can be dynamically composed. Resource adaptation can be achieved since every component is small and can be thrown away. Only those frequently used facets are cached to reduce the memory consumption. Functionality adaptation could be possible as we have the flexibility to retrieve necessary software components to carry out some unplanned functions/services according to the current context (e.g., location, friend nearby). To support the above adaptations, the facets are made up of two parts. More detailed discussions can be found in [7].

- *Shadow:* A RDF/OWL description of properties of the component including vendor, version, the functionality it fulfills, its dependencies and its resource requirements. The Sparkle proxy server will locate and select the appropriate facets on behalf of the requesting device according to the wanted functionality.
- *Code segment:* This is the body of the executable code, which implements the functionality. It follows the contract described in its shadow.

Figure 1 shows an example of facet dependency and a runtime snapshot of the *facet execution tree* formed by a calling sequence from T_0, T_1 to T_{10}. The dotted ellipse represents functionalities that compose a certain application. Each functionality has one or multiple black/white circles representing compatible facet implementations of the same functionality. Each functionality is associated with a context vector $CV =< C_{i1}, C_{i2}, .., C_{in} >$ at calling sequence ID i, where C_{in} means a context attribute relevant to this application. Note that facets of the same functionality may have different facet dependencies. For example, facet i of functionality A requires functionality D and E, while facet k requires functionality F and G. The tree formed by black

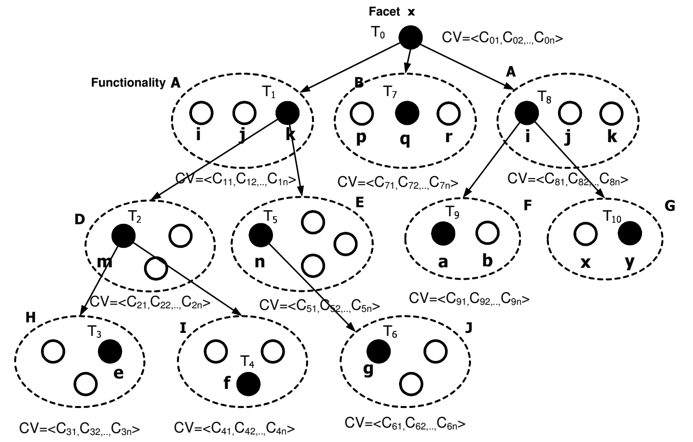


Fig. 1. Illustration of Facet Dependency with Calling Sequence and Facet Execution Tree

circles is the facet execution tree. Since the facet selected for each functionality may vary according to various contexts, the facet execution tree may be quite different for each user, under different circumstances.

In a pervasive computing environment, there could be a large number of components for different device configurations. Even for a specific device model, it is possible to use different versions of components to carry out the same functionality. For example, functionality A in Figure 1 has three compatible facet implementations (i, j, k). Thus, it is not possible for developers to create components that are suitable for all devices due to the large variety of components for different device configurations. Therefore, the responsibility to find and return a component suitable for the device configuration should not rest on the users. The network itself has to have some intelligence in returning a suitable component for the client. The network should be able to tailor-make its response according to the particular user needs and preferences. The response should also be efficient to support the high mobile nature of users. Thus, an intelligent proxy server is required, which will accept clients' requests, and respond to them efficiently according to the available runtime context information.

Figure 2 depicts the architecture of the component selection framework on Sparkle proxy server. It consists of seven entities. The *component agent* forwards the incoming request to the *facet manager*, which will do adaptive facet selection to find the most appropriate components. It is also responsible for updating the corresponding client's *facet cache*, so that the facet *keep/drop* instructions can be formed on behalf of clients. The reasoning engine in our design is embodied in the *adaptive facet selection* process, where the case-based reasoning techniques are applied to match similar situations when select a component for client application. The *context manager* collects all available context information relevant to the request to assist reasoning process. *Peer coordinator* is responsible for finding other proxy servers that are capable for locating the required components and context information

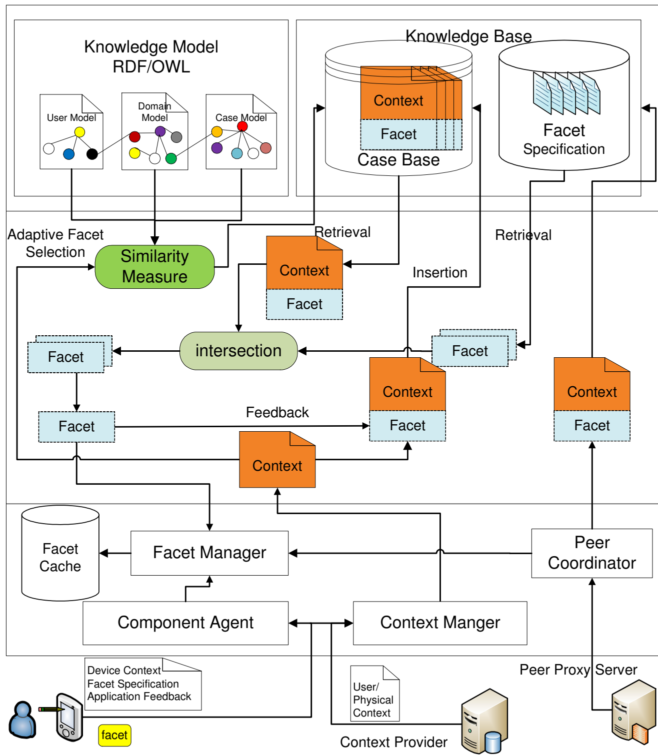


Fig. 2. Architecture of Intelligent Proxy Server in Sparkle

TABLE I
CONTEXT CATEGORIES USED FOR FACET/COMPONENT SELECTION

Category	Description	Example
Physical level context	information about computing resources and environment	memory size, network bandwidth
Application level context	data format and QoS requirement	image resolution
User level context	physio and phycological information of user	emotion, age, location

if necessary. *Knowledge model* and *knowledge base* contain domain knowledge and cases for the reasoning process.

IV. KNOWLEDGE REPRESENTATION

Web Ontology Language (OWL) enables the definition of domain concept and sharing domain vocabularies. The structure of a domain is described through classes and properties. In our component selection framework, domain knowledge and case structure are represented using OWL/RDF. We follow the methods discussed in CONON [16] for modeling context and supporting logic-based context reasoning. We have identified three categories of contexts that are taken into account for component selection in Table I.

We prototyped the context-aware personal communicator (CAPC) application. Its *physical level context* includes CPU, memory, network status. For *application level context*, it includes component cache status, data format requirement. For *user level context*, it includes user identity, age, occupation, social relationship, activity, location, mood. The case is not only used for knowledge storage but also for knowledge reasoning. The case representation is essential for every case-based reasoning system, especially for systems that are targeting to achieve context-awareness. All relevant contexts should be covered by the case representation. In our system, a "case" is defined as triple $Case = (Context, Solution, Result)$, where $Context$ is a description of current situation, represented as a *context vector* $< c_1, c_2, ..c_i, ..c_n >$, where c_i describes a context attribute which is relevant to the application. $Solution$ is a suggested facet (i.e., facet ID of the selected component) returned by the intelligent proxy using our CBR engine. The *Result* is the feedback of application execution from the client. The selection is done automatically as users may not have the expertise to select the right components. This also avoids the unnecessary conversations between applications and users to minimize intrusiveness. Thus, feedback from user and system is essential for the CBR system to ensure its selection accuracy and be able to evolve. We consider the system feedback by monitoring the component cache usage patterns and facet execution status. In the current design, a

```
<facet_case rdf:about = "#I1">
   <has_id rdf:datatype="&xsd;string">1</has_id>
   <has_time rdr:datatype="&xsd;dateTime">
            5/12/2008 10:30:52AM</has_time>
   <has_solutions ref:resource="#Solution1"/>
   <has_functionality rdf:datatype="&xsd;string">
            "Func1"</has_functionality>
   <has_location rdf:resource="#L1"/>
   <has_network rdf:resource="N1"/>
   <has_user rdf:resource="#P1"/>
   <has_dependency rdf:resource="#D1"/>
   <has_size rdf:resource="#S1"/>
   <has_device rdf:resource="#DVC1"/>
   <has_memoryuse rdf:resource="#MEMUS1"/>
</facet_case>

<solutions rdf:about="#Solution1">
   <rdf:Bag>
      <rdf:li>#Facet1</rdf:li>
      <rdf:li>#Used1</rdf:li>
   </rdf:Bag>
< /concurrence>

<dependency rdf:about="#D1">
   <rdf:Bag>
      <rdf:li>#FacetLevel2</rdf:li>
      <rdf:li>#Used5</rdf:li>
   </rdf:Bag>
</dependency>

<size rdf:about = "#S1">
   <has_static>2055</has_static>
   <has_dynamic>20567</has_dynamic>
   <has_guarantee>24543</has_guarantee>
   <has_unit>byte</has_uit>
</size>

<device rdf:about = "#DVC1">
   <has_id>1</has_id>
   <has_cpu ref:resource="#CPU1"/>
   <has_mem ref:resource="#MEM1"/>
</device>
```

Fig. 3. An RDF Presentation of a "Case"

case is represented as an OWL/RDF instance as shown in Figure 3. We store case instances into the relational database. Other advanced persistent storage can be deployed for storing the case instances. For example, D. Jeong et al. [17] provided a new persistent storage to efficiently manage OWL Web ontologies.

V. SIMILARITY MEASUREMENT

The core of a CBR system is the similarity measurement. We define a similarity function to find k past cases which are similar to the new case. The similarity between a new case N and a past case P can be calculated using Equation (1):

$$Similarity(\overrightarrow{N}, \overrightarrow{P}) = \frac{\sum_{i=1}^{n} Sim(N_i, P_i) \times W_i}{\sum_{i=1}^{n} W_i} \quad (1)$$

where N_i is the ith attribute of the new case N, P_i the ith attribute of the past case P, and n the number of attributes (relevant context) in the case. Sim(N_i,P_i) is the local similarity measurement between two attribute values. The $Similarity(\overrightarrow{N}, \overrightarrow{P})$ is the global similarity measurement

between the two cases, which is a weighted sum of all local similarity values. The weight W_i is in the range of [0,1]. The domain knowledge will be used to adjust the weight dynamically based on the inference rules stored in the knowledge base. We identify three types of similarity functions:

- **Numerical similarity**. Range similarity is applicable to numerical attributes (e.g. location, network). The similarity of two numerical attributes is defined as Equation (2):
$$Sim(N_i, P_i) = 1 - \frac{|N_i - P_i|}{interval} \quad (2)$$
where *interval* is a predefined threshold, when $|N_i - P_i| > interval$, Sim(N_i,P_i)=0, and therefore the range of Sim(N_i,P_i) is also [0,1].
- **Literal similarity**. Boolean similarity is applicable to literal attributes (e.g. gender, identity). The boolean similarity is defined as Equation (3):
$$Sim(N_i, P_i) = \begin{cases} 1 & N_i = P_i \\ 0 & otherwise \end{cases} \quad (3)$$
- **Ontological similarity**. Ontological similarity is applicable to the individuals of concept (e.g. occupation, activity). The closer are the two concepts in the concept hierarchy, the higher their similarity will be. We adopt the solution from jCOLIBRI2 [18], which provides four different functions for computing the concept-based similarity that depends on the location of the individuals in the ontology.

To enhance accuracy and personalization with context-awareness, we proposed a mechanism named *context-guided reasoning*. Currently we adopt a weight-assignment approach, in which the weight of context attributes in similarity measurement evolves along with the growth of the case base. We use the Equation (4) to generate the weight W_k for the kth context attribute which has discrete or enumerate values.

$$W_k = \sum p(c_k^j) \times \max(p(s_i|c_k^j)) \quad (4)$$

$p\left(c_k^j\right)$ is the probability when the kth context's value is c_k^j, $p\left(s_i|c_k^j\right)$ is the probability when the solution value is s_i with c_k^j as the kth context's value. We take the equation to measure how much the kth context will impact the solution. For context with continuous value, we do clustering operations on its value before we use the equation. To improve the reasoning speed, a *context-guided filtering* function was added. This mechanism can further reduce the number of candidate cases for similarity measurement. User's situation was estimated and only those cases under similar situation were fetched for reasoning.

VI. Adaptive Facet Selection

Core to realize context-awareness in Sparkle is to achieve *adaptive facet selection* collectively by proxy, client, and their interaction. The proxy server's *component agent* will run Algorithm 1 to selectively choose facets to be suggested to clients. A mobile client will also have an adaptive execution environment (a.k.a. extended facet manager) that will send/update context and application feedback to proxy server and get proxy's suggested facets and their keep/drop instructions. We extended our previous *facet execution tree* with an auxiliary data structure to store contexts as shown in Figure 1. Whenever the proxy server receives a facet request from client, its *component agent* will do case-based reasoning to find the most appropriate facet according to client's current reported contexts (Line 1 - 6). It also explores one more level of facet dependency (Line 10 - 23) to prefetch some facets to avoid future communications, which has a better user perceived experience than pure on-demand facet execution of applications. To accurately prefetch more relevant facets to be used in future, we set the context weighted preference vector to only include user level contexts (Line 15). Algorithm 2 will be mainly used to retrieve the most relevant cases in the knowledge base with a context filer to speed up searching performance. So far we implemented context filer as simple regular expressions on certain context attribute, whose value is usually known as a priori for an application in the *knowledge base*. In Algorithm 2, we use the *k-Nearest Neighbors* (KNN) algorithm to get most similar cases matched the given context. We select the *solution* with highest similarity value and the wanted function ID from these cases.

Algorithm 1 Case-based Facet Selection

Input: facet function ID $funcID$, context vector $\overrightarrow{CV}$, context filter cf, context weighted preference vector $\overrightarrow{W}$

1: create a temporary context filter $temp$ to match cases, whose contexts are associated with current facet execution tree T
2: **if** Match($funcId$, $\overrightarrow{CV}$, $temp$, $\overrightarrow{W}$) $!=$ **null then**
3:    Facet $f \Leftarrow$ the matched facet in T;
4: **else**
5:    Facet $f \Leftarrow$ Match($funcId$, $\overrightarrow{CV}$, cf, $\overrightarrow{W}$)
6: **end if**
7: $NeededFacets \Leftarrow$ the list of facet to prefetch in the *facet manager* of client
8: Add f to $NeededFacets$
9: $NonDrop \Leftarrow$ the list of facet to keep in *facet manager* of client
10: Get the shadow file of f
11: **for** each $funcId$ in the shadow's dependency **do**
12:    **if** $funcId$ is a pre-requisite or used in execution tree T **then**
13:       the dependent facet *df* $\Leftarrow$ Match($funcId$, $\overrightarrow{CV}$, cf, $\overrightarrow{W}$)
14:    **else**
15:       relax $\overrightarrow{W}$ to consider only *user level* context
16:       the dependent facet *df* $\Leftarrow$ Match($funcId$, $\overrightarrow{CV}$, cf, $\overrightarrow{W}$)
17:    **end if**
18:    **if** some facet of $funcId$ is in client's facet cache **then**
19:       add *df* to $NonDrop$
20:    **else**
21:       add *df* to $NeededFacets$
22:    **end if**
23: **end for**

The mobile client counterpart in Sparkle has an *extended facet manager* to run Algorithm 3 that interacts with Sparkle proxy servers during facet execution, by sending physical-level, application-level and user-level contexts to proxy servers. A typical process flow in client's runtime execution environment is as follows: the application on mobile

Algorithm 2 Match($funcId$, $\vec{CV}$, cf, $\vec{W}$)

Input: facet function ID $funcID$, context vector $\overrightarrow{CV}$, context filter cf, context weighted preference vector $\overrightarrow{W}$

1: **for** each context C_i of context vector $\overrightarrow{CV}$ **do**
2: determine similarity measurement equation to be applied to C_i according to its attribute type (numerical, literal, ontological)
3: **if** context C_i is included in $\overrightarrow{W}$ **then**
4: set the W_i to be the weight of C_i
5: **else**
6: use Equation (4) as weight of C_i
7: **end if**
8: **end for**
9: $Base \Leftarrow$ cases in CBR engine by context-guided filtering with cf
10: **for** each $Case_i$ in $Base$ **do**
11: calculate global similarity $Similarity(\overrightarrow{CV}, Case_i.context)$
12: **end for**
13: apply KNN algorithm to get the top 5 most similar cases
14: return the facet with function ID = $funcID$ and its facet ID = $solution$ of the most similar case among the 5 cases.

client will search a facet, which is triggered by user interaction or *facet dependency*; the adaptive facet manager will intercept application's facet request and search its local facet cache; if local facet cache does not have the facet with needed function ID, it will send a facet request with current contexts to proxy server's component agent; the component agent will return a facet list (for current on-demand execution or prefetching of dependent facets with one more level) and their *keep/drop* instructions. Upon receiving these information, the client will update its facet cache and *keep/drop* instructions, then continue its normal execution. If the client's system memory is below certain threshold, it will start to drop some facets in its facet cache, according to *keep/drop* instructions and the Least Recent Used (LRU) policy. Depending on whether exceptions happened in the facet execution tree rooted at current facet, a positive/negative feedback will be sent to proxy and a new case will be added into *knowledge base* of proxy server.

Algorithm 3 Execution Flow of Extended Facet Manager

1: **if** need to run next level facet **then**
2: get current context $\overrightarrow{CV}$ of all categories
3: send a facet request to proxy with current $\overrightarrow{CV}$
4: List $Nondrop$ = the returned list of facets to keep in *facet cache* from proxy
5: List $Neededfacets$ = the returned list of prefetched facets from proxy
6: Facet f = $Neededfacets$.first /*the first one is for current execution*/
7: **end if**
8: f.Execute();
9: $result \Leftarrow$ **True**
10: **if** Any exception is thrown during execution rooted at f **then**
11: $result \Leftarrow$ **False**
12: get current context $\overrightarrow{CV}$ of all categories
13: $Case \Leftarrow (\overrightarrow{CV}$, f, $result)$
14: notifyProxy($Case$)
15: **end if**
16: **if** system memory is below threshold s **then**
17: drop facet f that is least recently used (LRU) and is not in $Nondrop$
18: **end if**

VII. Evaluation

We implemented a prototype of the proposed distributed component selection framework in Sparkle proxy to prove feasibility and performance of using the case-based reasoning technique in component selection of mobile context-aware applications. We developed a client-side application, called *context-aware personal communicator* (CAPC) on an HP iPAQ H5500 PDA device with Pocket PC 2003. The Mysaifu JVM installed in the PDA supports Java serialization and reflection in a Windows Mobile environment. The intelligent proxy server was deployed on a desktop PC (OS: Windows Vista, CPU: Intel Core 2 Duo 3.0GHz). Current functions supported by the personal communicator are shown in Table II. Memory requirement of the personal communicator and more information about current facet implementation are given in Table III. To evaluate in a practical component-selection scenario, we analyzed the calling dependencies among functional components of an E-mail software used in the *personal communicator* with a Java code analysis tool [19]. For each functionality of this software, we implemented candidate facets by converting Java code from three E-mail packages: JavaMail [20], Mail4ME [21] and Tiger Jmail [22] to fulfill the requested functionality. Figure 4 shows part of the dependency relationships among several frequently used Java components in the E-mail program.

TABLE II
Major Functions of Personal Communicator

First-level Functional Units	Internal Functional Units
Online/offline message sending	Jabber message parser
E-mail	Roster list builder
Real-time chatting	File breaker
File delivering	File composer
Roster list adding/dropping	Roster entry finder
Regist/Unregist	Presence modifier

TABLE III
Implementation Details of Personal Communicator

Total no. of functions	70
Average no. of facets per function	5-6
Average size of a facet	3.6 KB
Total code size on client (w/ CAPC)	657 KB
Code size of Sparkle client	333 KB

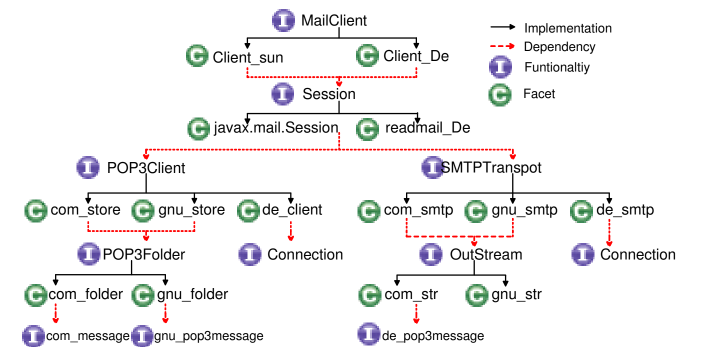
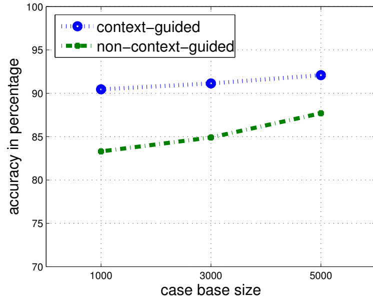
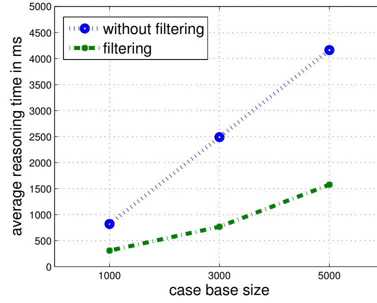


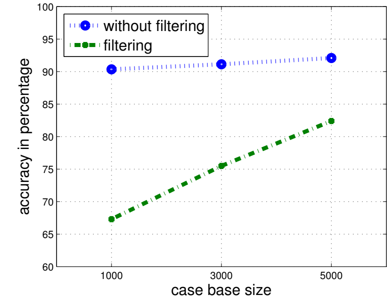
Fig. 4. Dependency Graph of Components Used in CAPC's E-mail Program



(a) Accuracy of Context-guided Reasoning



(b) Response Time of Context Filtering



(c) Accuracy of Context Filtering

Fig. 5. Performance of the CBR Engine

A. Evaluation of the reasoning engine

Currently, there are 26 context attributes used to describe all categories of contexts. Simulations were carried out with different case-base sizes (1000, 3000 and 5000) and we implemented the match function described in Algorithm 2 with jCOLIBRI2 [18] *NNretrieval similarity functions* to reason and select solutions. We use a two-fold cross-validation method [23] to evaluate the accuracy of the reasoner. Based on Figure 4, we create an execution trace to simulate the component selection process under various contexts. To obtain enough cases under many different contexts, users' behaviors (i.e., functions required under certain situations) were artificially produced by a set of pre-defined rules. Results and analysis of the simulation are discussed as follows. The measured response time did not include network latency as we only concern the speed of the reasoning engine and the result accuracy.

Figure 5 (a) shows the accuracy improvement with respect to the case base size. Whether the context-guided mechanism is applied or not, accuracy in both cases is satisfactory (at least 83%). The context-guided reasoning can further improve accuracy about 10%, as the dynamically assigned weight for context attribute can help find out more relevance cases.

Figure 5 (b) shows the reasoning time with or without the filtering function. Without the filtering function, the reasoning time could grow as large as 4 seconds when the case base has 5000 candidates. The filtering function could effectively reduce the processing time. A short response time of 1.5 seconds could be achieved in the same case. This method can obtain significant speed-up on reasoning process. However, the filtering method might put side-effect on accuracy. Figure 5 (c) shows the comparison of accuracy achieved in two reasoning processes. From the figure, we suspect the filtering method might ignore some useful cases, which caused accuracy to decline. This was due to our filtering rules that are too static. Yet we have saved about 65% reasoning time in the case of large case size (e.g. at 5000) at a loss of less than 10% accuracy. In future work, we should further investigate the filtering mechanism to make it more reliable.

B. Evaluation of the system performance

To show how the case-based component selection framework can actually improve applications' performance, we measured system performances during a user's interaction with the E-mail program. We simulated various contexts with limited computation resources. The framework selected most suitable components according to the circumstances to achieve most economical memory usage, shortest execution time of user actions and to avoid execution failures. The parameter settings of the evaluation are given in Table IV.

TABLE IV
PARAMETER SETTINGS OF EXPERIMENT

Total no. of emails to read	50
Average size of an email to read	0.7 MB
Total no. of emails to send	25
Average size of an email to send	1.4 MB (with file attachment)
JVM memory limit	3072 KB - 6144 KB
Network bandwidth	20 KBps
Total no. of facets	15

We first recorded a typical execution sequence of a user, then executed the same sequence on four different schemes: (1) pure JavaMail implementation; (2) pure Mail4ME implementation; (3) pure Tiger Jmail implementation; (4) our component selection. We adopted java soft reference to clear useless facets. The memory consumption of each implementation at each execution step is shown as Figure 6. The result of our approach is best in memory usage. Its maximum memory consumption is reduced by more than 20%, since only facets with smallest memory consumption were selected during reasoning process.

Table V compares the execution time of various E-mail schemes at different stages of execution. The proposed solution can speedup the application initialization and message reading/sending operations. Though our implementation can not outperform stand-alone E-mail programs in some performance metrics, its total execution time is the shortest.

Overall, we have demonstrated the flexibility of our component selection framework in adapting the context changes, including user-level functional requirements and low-level memory and network availability. The integration of the case-based reasoning solution and the dynamic component-based

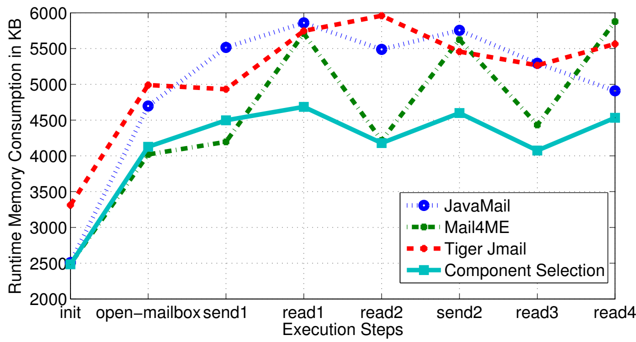


Fig. 6. Evaluation of Sparkle Memory Management

TABLE V
PERFORMANCE COMPARISONS WITH JAVAMAIL, MAIL4ME AND TIGER

Performance Item	JavaMail	Mail4ME	Tiger Jmail	Our Solution
Time of initializing and opening mailbox	48 s	9 s	50 s	10 s
Average message reading time	156 ms	235 ms	328 ms	240 ms
Average message sending time	29 s	41 s (failed)	30 s	30 s
Total execution time: 50 emails read, 25 sent	789 s	1286 s	832 s	776 s
Maximum memory consumption (in MB)	5.8	5.9	6.0	4.5

composition mechanism made it possible to achieve light-weight context-aware computing in a small device. Potentially, unlimited functionalities can be supported at runtime to meet users' needs.

VIII. CONCLUSION AND FUTURE WORK

The proliferation of pervasive computing is fostering the need for applications to provide more personalization and context-awareness. In this work, we presented the main concept behind our distributed component selection framework. We use a case-based reasoning technique to provide proactive component selection based on context information. Context-awareness and personalization are embodied in the reasoning and selection process.Resource adaptation and functionality adaptation are enabled as the component selection could be done according to the current context. Various experiments have been conducted to show the feasibility of using case-based reasoning technique in mobile context-aware applications. In our future work, various optimizations will be studied to further improve the system performance. For example, using index with context, reasoning efficiency could be improved; clustering similar cases could shrink the case base size; and seed case selection could help to generate typical cases. Besides, we will refine our context-guided reasoning and filtering mechanism to provide more rational and reliable functionalities. To serve users proactively, user's intention model is needed to consolidate our knowledge-intensive reasoning.

REFERENCES

[1] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, October 2002.

[2] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "Pcom - a component system for pervasive computing," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*, 2004, pp. 67–76.

[3] M. Satyanarayanan, "Pervasive computing: vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.

[4] T. Ma, Y.-D. Kim, Q. Ma, M. Tang, and W. Zhou, "Context-aware implementation based on cbr for smart home," in *IEEE International Conference on Wireless And Mobile Computing, Networking And Communications*, vol. 4, 2005, pp. 112–115.

[5] A. Kofod-Petersen and A. Aamodt, "Contextualised ambient intelligence through case-based reasoning," in *Proceedings of the Eighth European Conference on Case-Based Reasoning*, vol. 4106, Ölüdeniz, Turkey, September 2006, pp. 211–225.

[6] M. J. Khan, M. M. Awais, and S. Shamail, "Self-configuration in autonomic systems using clustered cbr approach," in *International Conference on Autonomic Computing*, 2008, pp. 211–212.

[7] N. M. Belaramani, Y. Chow, V. W. man Kwan, C.-L. Wang, and F. C. Lau, "A component-based software architecture for pervasive computing," in *Technologies and Applications in Distributed Virtual Environments, World Scientific Publishing Co*, 2003, pp. 201–222.

[8] J. Lee and J. Lee, "Context awareness by case-based reasoning in a music recommendation system," *Ubiquitous Computing Systems*, pp. 45–58, 2008.

[9] D. Zheng, Y. Jia, P. Zhou, and W.-H. Han, "Context-aware middleware support for component based applications in pervasive computing," in *Advanced Parallel Processing Technologies*, 2007, pp. 161–171.

[10] D. Ayed, C. Taconet, G. Bernard, and Y. Berbers, "An adaptation methodology for the deployment of mobile component-based applications," in *ACS/IEEE International Conference on Pervasive Services*, 2006, pp. 193–202.

[11] A. K. Petersen, "Challenges in case-based reasoning for context awareness in ambient intelligent systems," in *Workshop Proceedings of 8th European Conference on Case-Based Reasoning*, Ölüdeniz/Fethiye, Turkey, September 2006, pp. 287–299.

[12] T. C. Lech and L. W. M. Wienhofen, "Ambieagents: a scalable infrastructure for mobile and context-aware information services," in *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, USA, 2005, pp. 625–631.

[13] A. Aamodt and E. Plaza, "Case-based reasoning: foundational issues, methodological variations, and system approaches," *AI Commun.*, vol. 7, no. 1, pp. 39–59, March 1994.

[14] A. K. Petersen and A. Aamodt, "Case-based situation assessment in a mobile context-aware system," in *Artificial Intelligence in Mobile Systems (AIMS 2003), Universität des Saarlandes*, 2003, pp. 41–49.

[15] N. Dhanakoti, S. Gopalan, V. Sridhar, and S. Subramani, "A distributed service discovery and selection framework in pervasive service environments," in *Proceedings of Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference*, 2005, pp. 452–457.

[16] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, "Ontology based context modeling and reasoning using owl," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, 2004, pp. 18–22.

[17] D. Jeong, M. Choi, Y.-S. Jeon, Y.-H. Han, L. Yang, Y.-S. Jeong, and S.-K. Han, "Persistent storage system for efficient management of owl web ontology," *Ubiquitous Intelligence and Computing*, pp. 1089–1097, 2007.

[18] J. A. Recio-García, B. Díaz-Agudo, P. A. González-Calero, and A. Sánchez-Ruiz-Granados, *Ontology based CBR with jCOLIBRI*, R. Ellis, T. Allen, and A. Tuson, Eds. Cambridge, United Kingdom: Springer, December 2006.

[19] M. M. Ivica Aracic, "Flexible abstraction techniques for graph-based visualizations," in *Eclipse Technology eXchange workshop (eTX) at ECOOP*, 2006.

[20] *Sun Microsystems, Inc API Java Mail: http://java.sun.com/products/javamail/*.

[21] *Enhyndra.org, Mail4ME API: http://mail4me.objectweb.org/*.

[22] *Tiger Privacy, Tiger Jmail API: http://tjmail.sourceforge.net/*.

[23] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *International Joint Conference on Artificial Intelligence*, 1995, pp. 1137–1145.