Q-CAD: QoS and Context Aware Discovery Framework for Adaptive Mobile Systems

Licia Capra, Stefanos Zachariadis and Cecilia Mascolo
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{L.Capra|S.Zachariadis|C.Mascolo}@cs.ucl.ac.uk

Abstract

Pervasive computing environments are populated by a large number of heterogeneous and dynamic resources, encompassing devices, services and information sources. This number is set to radically increase in the future; as a result, a mobile device will be able to contact a large number of service providers and sensors that will enable it to perform any task at hand. Moreover, interaction with these services and sensors will be made possible by means of various components, some located on the mobile device, some available for download from remote hosts. We refer to services, sensors and components as resources. In this paper we present Q-CAD, a resource discovery framework that enables pervasive computing applications to discover and select the resource(s) best satisfying the user needs, taking the current execution context and quality-of-service (QoS) requirements into account. The available resources are screened, so that only those suitable to the current execution context of the application will be considered; the shortlisted resources are then evaluated against the QoS needs of the application, and a binding is established to the best available. The paper illustrates how we encode context and QoS information, gives details of the Q-CAD model and of its mapping onto a component-based architecture and finally reports on the implementation and experimental results.

1 Introduction

Technological advances, both in wireless networking and portable device capabilities, have met social popularity, so that we are now witnessing an increase in the number of devices and services we use to accomplish our daily tasks. It will not be long before the commercially exploitable potentials of these technologies will be apparent, resulting in a large number of interconnected devices, provided services and publicly available information sources. Interaction with these services and devices will be enabled by means of various components, some located on the mobile device, some available for download remotely. We refer to these services, devices and components as *resources*. However, in order to bring Mark Weiser's vision of ubiquitous computing into reality, more efforts are needed to minimise the disruption of the user in managing these resources, while maximising his/her satisfaction at the same time.

Research in the area of resource discovery for pervasive environments has been very intense in recent years. Its main focus has been on the development of efficient algorithms that take the network topology of pervasive scenarios into account when routing advertisements and queries (e.g., [15][14]). However, most of these approaches do not consider user preferences. We argue that, in order to improve the user experience, resource discovery and selection protocols for pervasive environments should be able to discover and bind to the resources that the user considers most suited in the current execution context and according to his/her quality-of-service (QoS) needs. By context, we refer to any piece of information that is of interest to the execution of the application itself. This definition includes both information that is local to a device, such as available battery and memory, and information that is external to the device, such as services and other devices in reach. With QoS needs, we refer to any non-functional requirement the user may express (i.e., proximity of a service provider, cost of a service, etc).

In this paper we present Q-CAD, a *context* and *QoS aware* resource discovery and selection framework for pervasive environments. Each application encodes in an *application profile* the way context should influence the discovery of, and the binding to, resources; Q-CAD uses this information to reduce the resources available to the application in the current context to a subset of 'plausible' ones. Each application also encodes the QoS needs of the user into a *utility function* that Q-CAD applies to select the most suitable resource (i.e., the one that maximises the utility of the user) among the plausible ones. To cater for the varying context and non-functional requirements of the user, both the application profile and utility function can be changed dynamically.

The paper is structured as follows: in Section 2 we introduce a scenario that highlights the major goals of this work and summarize its assumptions. Section 3 describes Q-CAD application profiles and utility functions, and details the discovery and selection protocol. In Section 4 we present the Q-CAD architecture and in Section 5 we discuss the Q-CAD performance evaluation. Section 6 compares Q-CAD with other work in the field. Finally, Section 7 concludes the paper and lists some future work.

2 Case Study

Let us consider two scenarios of discovery: in the first, a tourist named Alice is on holiday in New York and wishes to print the pictures she has taken with her digital camera. In order to do so, she has to discover and select a photo development service provider, among the many available. Different parameters may influence this choice: for example, location of the provider (Alice may prefer a provider located close to her hotel, as to be able to collect the prints conveniently), cost of the service, quality of the prints, and so on. In the second scenario, we imagine that a number of sensors[1] have been deployed in the most prominent locations of the city, providing tourist information to other devices in proximity. While on a bus tour, Alice may use her PDA to dynamically discover and bind to these sensors; however, different tourist companies may have deployed their own, providing different quality/amount of information for different prices to potential customers. The non-functional requirements of Alice must thus be used to decide what sensor to bind to amongst the discovered ones. Context must be taken into account too, as, for example, audio information may be preferred to both audio and video in low energy conditions. Moreover, the processing of the data may require software components that are not installed on Alice's PDA, so that dynamic discovery, download and deployment of components from available repositories can become part of the process too. Note that, although fictitious, these scenarios are not too far from reality, as shown by the Urban Tapestry project [22].

We refer to the first scenario as *proactive discovery*: the discovery and binding to a service provider is a consequence of an explicit request of the user to locate such a service. The second scenario is an instance of *reactive discovery* instead: the discovery and binding to sensors and component repositories is a result of context changes. Although discovery is triggered by different events in the two scenarios (user action in the first scenario, context change in the second scenario), both types of discovery demand a similar behaviour from the discovery framework: locating and binding to a resource (be it a service provider, a sensor, or a component) that is best suited in the current context (*context-awareness*) and according to the current non-functional requirements of the user (*QoS-awareness*).

In designing the Q-CAD discovery model for pervasive computing applications, we thus aimed to: (1) provide applications with a means to explicitly state the context conditions of interest to their user (i.e., context awareness); (2) provide applications with a means to explicitly state the non-functional requirements of the user (i.e., QoS awareness); (3) develop a resource discovery and selection protocol that takes the preferences of the user into account (both in terms of context and of QoS needs).

Q-CAD builds on the following assumptions: the existence of a shared ontology to refer to context elements and conditions, resource names and characteristics, and non-functional requirements; the integration with an existing discovery protocol for pervasive networks on which Q-CAD relies to route advertisements and queries (in this paper, we do not bind to any in particular); finally, the usage of a local component model to represent applications.

3 Q-CAD Model

Q-CAD achieves context and QoS awareness by means of *application profiles* and *utility functions* respectively. In this section, we first describe the information they encode and then illustrate how the discovery and selection protocol uses them. In our implementation of the Q-CAD discovery model, we have chosen to encode both application profiles and utility functions using XML; in the following, we illustrate examples written in XML to ease presentation, even though the actual language used is not fundamental to the model. The complete XML Schema specifications we defined are available online [7].

3.1 Q-CAD Resources, Descriptors and Binding

Central to our model is the notion of a *resource*. Before detailing what information is encoded in application profiles and utility functions, we define what a resource is in this setting, what *binding* to a resource implies and we introduce the concept of *resource descriptor*.

The resources that the Q-CAD model considers are: *services* provided by remote providers, *sensors* from which an application may get data, and *components* located remotely and that can be downloaded and deployed on the local host.

[1] We generally call *sensor* a device with limited resources and whose only task is to sense the environment.

```
(component, displayVideo)
(code, display800600.jar)
(resolution, 800x600)
(version, 2.1)
(platform, JVM2)
(size, 70KB)
(cost, $10)
(memory, 2)
(battery, 4)
```

Figure 1. Example of Resource Descriptor.

We refer to these resources as *remote resources*, to distinguish them from those local to a device (e.g., battery, memory, CPU, etc.). We assume remote resources are uniquely identified by means of an addressable naming scheme that is resolved by the underlying communication framework. The namespace used can be local, or global, although we expect that, in practice, a combination of the two will be used. For example, considering a device with both cellular and ad-hoc Bluetooth interface, a global naming scheme can be used for the cellular interface, while a local one can be used for the Bluetooth interface. In our implementation, we associated a unique Uniform Resource Identifier (URI) to each remote resource (e.g., a sensor can have `//machine//sensor0` URI). We define the *binding* to a resource (i.e., the last step of a resource discovery and selection process) as the association of the selected remote resource to a *component* that is local to the device and that is able to interact with it. A remote resource could itself be a component: in this case, binding refers to downloading and deploying the component on the local system.

Besides its URI, every remote resource is associated with a static specification, or *resource descriptor*, that characterises the resource by means of a list of attribute/value pairs. Figure 1 illustrates an example of a remote resource descriptor for a component that displays video (attribute `component`) at a resolution of $800x600$ (attribute `resolution`); information about the component implementation follows (e.g., version number, platform required, size of the component, etc.). In addition, the descriptor contains information that can be used to assess the quality of the resource itself; this includes, for example, estimates of local resources (e.g., battery and memory) consumption (these estimates vary in a range $[0, 10]$ in the descriptor shown in Figure 1). Note that these values do not aim to be precise estimates of actual consumptions; rather, they aim to enable comparisons of resources of the same type (e.g., services, components, etc.). In this paper, we trust that the estimates have not been maliciously altered.

3.2 Application Profiles

Application profiles specify how the user wishes the context to influence the discovery of remote resources, both in proactive and in reactive situations.

Proactive Discovery. For each remote resource the application may be willing to bind to, the proactive encoding of its profile contains an association between the resource name (tag `<BIND_RESOURCE>`) and the context conditions that must hold for the binding to be enabled (tag `<REMOTE_CONTEXT>`). For example, the encoding shown in Figure 2 states that only printing service providers that give customers at least 100MB of disk space should be considered. This condition acts as a filter over the possibly high number of providers of the same service. Only one context configuration (tag `<REMOTE_CONTEXT id="1">`), containing a single condition (tag `<CONDITION>`) is specified. More generally, multiple contexts can be associated to the same binding resource, and more conditions may be associated to the same context; for example, another condition could be to consider only service providers with load lower than a threshold. The semantics of these encodings are the following: the binding to the remote resource is enabled if and only if *at least* one of the context configurations is enabled (*or* semantics); a context configuration is enabled if and only if *all* the conditions associated to it hold (*and* semantics). If more than one service provider passes the filtering, the actual provider to bind to will be selected using the application's utility function (see Section 3.3). As Figure 2 shows, the proactive part of the profile enables richer encodings than the one illustrated so far, to further support dynamic adaptation to context. In particular, the application may specify in which contexts (initial tags `<LOCAL_CONTEXT>` and `<REMOTE_CONTEXT>`) the discovery and binding process should be enabled; for example, it may be forbidden when energy is low (local condition), or when the network connection is too unstable (remote condition). In our example, these contexts are not specified, thus indicating no pre-condition to the discovery and binding process.

Once a remote service provider has been discovered and selected, the application has to decide how to interact with it, as different behaviours/protocols may be available. We call *binding* the last step of the resource discovery process that associates the remote service provider to the local component that implements the desired behaviour/protocol. The component should be selected out of a list of desirable ones (tag `<ADAPT_COMPONENT>`); the choice depends on the following information, that is attached to each of these components: local context (tag `<LOCAL_CONTEXT>`), remote context (tag `<REMOTE_CONTEXT>`), and application preferences (tag `<ATTRIBUTES>`). For example, the encoding of Figure 2 dictates that pictures should be uploaded to the provider site using a component that supports an encryption protocol when the remaining battery is above 30%, while using a plaintext upload otherwise. As we will discuss later, application preferences can be evaluated by comparing the values of the attributes listed in the profile (i.e.,

```
<PROACTIVE id="1">
  <LOCAL_CONTEXT/>
  <REMOTE_CONTEXT/>
  <BIND>
    <BIND_RESOURCE name="printPicture">
      <REMOTE_CONTEXT id="1">
        <CONDITION name="diskSpace" op="greaterThan" value="100MB"/>
      </REMOTE_CONTEXT>
    </BIND_RESOURCE>
  </BIND>

  <ADAPT>
    <ADAPT_COMPONENT id="1">
      <LOCAL_CONTEXT id="2">
        <CONDITION name="battery" op="greaterThan" value="30%"/>
      </LOCAL_CONTEXT>
      <REMOTE_CONTEXT/>
      <ATTRIBUTES>
        <ATTRIBUTE key="component" op="equals" value="encryptedUpload"/>
      </ATTRIBUTES>
    </ADAPT_COMPONENT>

    <ADAPT_COMPONENT id="2">
      <LOCAL_CONTEXT id="3">
        <CONDITION name="battery" op="lessThan" value="30%"/>
      </LOCAL_CONTEXT>
      <REMOTE_CONTEXT/>
      <ATTRIBUTES>
         <ATTRIBUTE key="component" op="equals" value="plaintextUpload"/>
         <ATTRIBUTE key="location" op="equals" value="local"/>
      </ATTRIBUTES>
    </ADAPT_COMPONENT>
  </ADAPT>
</PROACTIVE>
```

Figure 2. Example of Proactive Encoding.

```
<REACTIVE id="1">
  <LOCAL_CONTEXT id="1">
    <CONDITION name="battery" op="greaterThan" value="30%"/>
  </LOCAL_CONTEXT>

  <REMOTE_CONTEXT id="2">
    <ATTRIBUTES>
      <ATTRIBUTE key="sensor" op="equals" value="videoSensor"/>
      <ATTRIBUTE key="resolution" op="equal" value="800x600"/>
      <ATTRIBUTE key="format" op="equals" value="jpeg"/>
    </ATTRIBUTES>
  </REMOTE_CONTEXT>

  <BIND>
    <BIND_RESOURCE name="videoSensor"/>
  </BIND>

  <ADAPT>
    <ADAPT_COMPONENT id="3">
       <LOCAL_CONTEXT/>
       <REMOTE_CONTEXT/>

       <ATTRIBUTES>
         <ATTRIBUTE key="component" op="equals" value="displayVideo"/>
         <ATTRIBUTE key="cache" op="greaterThan" value="1024KB"/>
         <ATTRIBUTE key="resolution" op="greaterThan" value="800x600"/>
       </ATTRIBUTES>
    </ADAPT_COMPONENT>
  </ADAPT>
</REACTIVE>
```

Figure 3. Example of Reactive Encoding.

`<ATTRIBUTE key=.../>`) with those that appear in the resource descriptors. More generally, in the `<ADAPT>` part of the application profile we may find: nothing, indicating that whatever component is able to support interaction with the remote resource will be used; one single component, without any context associated, thus requiring exactly a component that satisfies the given attributes (e.g., a component that implements an encrypted upload, regardless of context) to be used; finally, a list of components with associated attributes and contexts. If multiple components match the criteria given, the utility function will be used to select the one that best satisfies the QoS needs of the user (see Section 3.3). Note that the chosen component may not be available locally (e.g., it has not been loaded before due to memory limitations); in this case, discovery, download and deployment of a component implementation is required; as we will illustrate for reactive discovery, this process is almost identical to the one that has been discussed above, as components are treated as yet another type of resource. We can allow the download of components from specific (i.e., trusted) sites only, by means of the attribute `location`; if the value of this attribute is set to `local`, as in Figure 2, only components that are available locally can be used.

Reactive Discovery. The reactive part of the profile describes how the application reacts to context changes. The reactive encoding shown in Figure 3 states that, when the remaining battery power is greater than 30% (`<LOCAL_CONTEXT>`) and there is a video sensor in reach that broadcasts images at a resolution of 800x600 in the JPEG format (`<REMOTE_CONTEXT>`), a binding to that sensor should be established (tag `<BIND>`). As the example shows, context can be composed of both local resources (e.g., battery) and remote resources (e.g., video sensor); we use the tag `<CONDITION>` for the former, and `<ATTRIBUTES>` for the latter. Continuing with the example, after binding to a video sensor, the data received should be displayed on the local device using a component (tag `<ADAPT_COMPONENT>`) that can display images at the specified resolution, and that can cache at least 1024KB of the data received; once again, if a local implementation of that component is not available, one has to be discovered that satisfies the listed conditions. If, after the screening performed using context conditions, there are still multiple video sensors we may bind to, or multiple implementations of the desired component, the utility function selects the one to be used (see Section 3.3).

Let us now step back from the specific examples and summarise the information encoded in an application profile. The **first part** specifies the initial local and remote context pre-conditions to perform discovery and adaptation. Context can be composed of local resources (e.g., memory, battery, CPU, etc.), and remote resources (e.g., video sensor). The conditions associated to a remote resource (tags `<ATTRIBUTE>`) are used during the discovery of the remote resource itself, to cut down the number of suitable answers. When at least one local context *and* one remote context are enabled, the pre-condition to discovery and adaptation holds and a binding to a remote resource may be required. Note that this ‘context change triggers binding’ type of behaviour represents the very nature of reactive adaptation, that demands monitoring of context and prompt reaction to changes; for proactive adaptation, instead, these general local and remote context specifications will typically be left blank, and context conditions will rather be associated to specific bindings (second part), so to be evaluated only on-demand, when resource discovery is explicitly triggered. The **second part** specifies what bindings are necessary (either as a consequence of context change, or as a result of an application service request), and what context information should be used to reduce the number of plausible resources to bind to. Reactive encoding will typically require a binding to the very same resource discovered during the pre-condition, thus leaving the context associated to the binding empty; proactive encoding will specify here the context conditions necessary to prune the binding instead. The **third part** specifies what adaptation is required on the device itself, in order to bind to the selected remote resource. Each adaptation alternative may have a context associated to it, to state what alternative is most suited in different contexts.

The amount and complexity of information encoded in application profiles may seem to prescribe implementations that are too heavy for portable devices. We discuss in Section 4 and 5 how the Q-CAD architecture makes smart use of this information so that, for profiles of plausible complexity, the computational overhead is low.

3.3 Utility Functions

Application profiles enable context-aware discovery; however, more than one resource (be it a service provider, a component implementation or a sensor) may be suitable in the current context. Utility functions are used to select the best resource out of the context-suitable ones, according to the non-functional requirements of the user. Similarly to profiles, a utility function exists for each application, so that user preferences may vary depending on the particular application.

Let us suppose that we are looking for a component implementation to be downloaded and executed on our device, in order to interact with a remote sensor. Figure 4 illustrates an example of a utility function encoding. As shown, the encoding is divided into two parts: a `<MAXIMISE>` part, and a `<RETURN>` part. Under the tag `<MAXIMISE>`, the application lists the non-functional parameters it is interested in, together with weights that express their relative importance. Let us assume that these weights vary in the range $[0, 10]$; the example indicates that the application is

```
<UTILITY_FUNCTION id="uf1">
  <RETURN>
    <EVALUATE>
      <ATTRIBUTE key="cost" op="greaterThan" value="10$"/>
    </EVALUATE>
    <FILTER>
      <ATTRIBUTE key="cost"/>
    </FILTER>
  </RETURN>
  <MAXIMISE>
    <ATTRIBUTE key="battery" weight="10"/>
    <ATTRIBUTE key="memory" weight="5"/>
  </MAXIMISE>
</UTILITY_FUNCTION>
```

Figure 4. Example of a Utility Function.

willing to select a component that maximises battery and memory saving; also, saving energy is twice as important as saving memory. The `<MAXIMISE>` part of the utility function is executed on a resource descriptor, as a summation of products (i.e., normalised estimates multiplied by weights, as found in the resource descriptor and utility function, respectively); it returns a single value that can be used to compare the quality of different resources. Note that high weights associated to parameters such as battery and memory mean that the user aims at preserving them; however, resource descriptors estimate their consumption, rather than their preservation. In order to give higher scores to the remote resources that reduce consumption, we therefore use the value: saving = maximum consumption - estimated consumption. The resource discovery concludes with the selection of the resource that scored highest. However, there are cases in which we do not want the selection process to be fully automated. For example, we may not want to download a component that maximises our non-functional requirements, if it is too expensive. We use the `<RETURN>` part of the utility function specification when intervention on behalf of the application or user is required. For example, Figure 4 dictates that discovery and selection can be automated if the cost of the component is less than $\$10$; otherwise, information has to be prompted to the application to make the final decision. This information includes, besides the result of the maximisation part, all the attributes listed in the `<FILTER>` part of the utility function (these attributes are a subset of those that appear in the resource descriptor, and usually coincide with the ones used in the `<EVALUATE>` part).

3.4 Discovery Protocol

In this section, we present a conceptual description of the discovery protocol that Q-CAD adopts in order to achieve QoS and context awareness, based on application profiles, utility functions and resource descriptors. As shown in Figure 5, the Q-CAD discovery protocol consists of three main steps: *matching*, *evaluation* and *selection*. These steps are exactly the same, regardless of the type of remote resource sought, and of whether we are performing a reactive or proactive search (the Q-CAD architecture will take care of this aspect, instead, as discussed later on).

Matching. The first step of the protocol uses the information encoded in the application profile to perform context-aware resource discovery. On behalf of the application, Q-CAD sends a discovery message containing details about the wanted resource (e.g., component type, resolution, platform, etc.). This information can be found in the application profile and is used to prune the number of potential matches. Figure 6 illustrates an example of a discovery message associated to the reactive encoding of Figure 3;

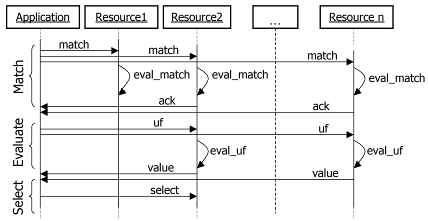

Figure 5. Three-Step Discovery Protocol.

as shown, we are looking for a component that displays images at a resolution of 800x600 or higher, and that can cache 1024KB of the images received. The remote resources receiving this message evaluate it locally against their resource descriptors, and only those matching the query will reply (resources 2 to n in Figure 5). When composing a discovery message to locate a service provider, we also specify the protocols the service provider must be able to speak (e.g., for the printing service of Figure 2, we would require `plaintextUpload` and `encryptedUpload` to be implemented by the server).

Evaluation. Once the replies of the matching phase have been received, the second step of the protocol uses the information encoded in the utility function to perform QoS evaluation. The resources that have survived the pruning receive a message containing the application's utility function. Each remote resource evaluates the function over the relevant resource descriptors and returns an answer to the querying application. Note that a resource may refuse to perform this computation, either because it does not have the capabilities to do so (as could be the case for a sensor), or because it does not want to consume local resources. On the other hand, the application may not be willing, for privacy reasons, to disclose its utility function. In these cases, the resource descriptor may be returned instead, and the application itself will compute the utility function over the descriptor locally.

Selection. If no application intervention is required, the resource that maximises the application utility is automatically selected, based on the answers received and/or the local computation performed; if intervention is required instead, the returned values are passed to the application to obtain a final choice. Once the selection has been made, the protocol concludes.

The full potential of the language used to encode application profiles allows for a cascading execution of the discovery protocol: for example, to bind to an arbitrary number of sensors that are relevant to the application, then to find a service provider that can process the data coming from the sensors, and finally to locate and download the components needed to talk to the sensors and service provider. Although possible in principle, this situation is far from any realistic scenario we have experimented with. Furthermore, we found the profiles illustrated in Section 3.2 to be already expressive and representatives of most realistic situations. In these cases, the discovery protocol is repeated at most twice: first to discover a service provider or sensor, and then to discover a component to talk to it. In the following sections, we analyse how the Q-CAD model has been mapped onto an efficient architecture and implementation.

```
<MATCH>
  <ATTRIBUTE key="component" op="equals" value="displayVideo"/>
  <ATTRIBUTE key="cache" op="greaterThan" value="1024KB"/>
  <ATTRIBUTE key="resolution" op="greaterThan" value="800x600"/>
</MATCH>
```

Figure 6. Example of a Discovery Message.

4 Q-CAD Architecture

The Q-CAD architecture is organised into four conceptual layers: the Application Meta-Interface layer, the Information layer, the Decision layer and the Action layer. As shown in Figure 7, these layers sit between the Application and the Communication layers. We assume applications are defined as locally interconnected components; component models geared for mobile devices, such as Beanome [8], Gravity [9] and SATIN [24] could be used for this purpose. Also, we rely on the routing capabilities of the Communication layer to route advertisements and queries. The Q-CAD architecture is further refined in Figure 8 as a collection of interacting components. We now describe each individual layer, the interaction between the components, and discuss issues related to their instantiations.

The Application Meta-Interface Layer. This layer encapsulates the interaction of the applications with the Q-CAD architecture. It is composed of the *PolicyRepo* and *NotificationService* components; at application start-up, an instance of these components is created and associated with it. PolicyRepo represents the *reflective* aspects of the application, as it allows for the dynamic inspection and modification of the application profile and utility function. The NotificationService is responsible for: extracting the information from the application profile and passing it on to the

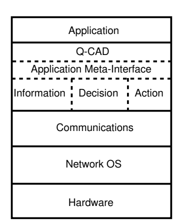

Figure 7. A High-Level Overview of the Q-CAD Architecture.

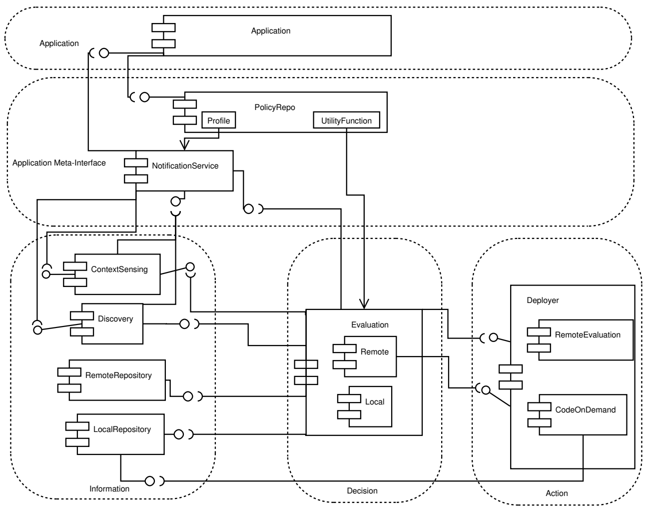

Figure 8. The Q-CAD Architecture.

components in the Information layer; notifying the Information layer of changes that happen in the profile via the PolicyRepo component; returning the result of a resource discovery to the application.

The Information Layer. This layer is responsible for the management of local and remote context-related information. Context information is used to know what part of the system must be monitored (either continuously, during reactive adaptation, or on-demand, during proactive adaptation); also, it is used during the matching part of the discovery and selection protocol to prune the number of potential matches for resource binding. The Information layer takes care of all these tasks. In particular, the *ContextSensing* component is responsible for monitoring the state of the local system (e.g., available memory, remaining battery power, etc.), while the *Discovery* component is responsible for detecting the remote resources (in particular, services and sensors) currently available to the local host, that the application is interested in. Together, these components are thus responsible for the conditions set in the various `<LOCAL_CONTEXT>` and `<REMOTE_CONTEXT>` elements of the profile. Note that the realisation of these components may require the instantiation of multiple (sub)components, each monitoring different parts of the context, using different techniques such as polling and interrupts. To preserve local resources, only conditions expressed in the profiles of running applications are monitored; moreover, computationally inexpensive conditions (e.g., remaining battery power) are checked first and only when these are satisfied will more expensive conditions (e.g., existence of a remote sensor) be monitored (*and* semantics of context conditions). Note also that it is the responsibility of the Information layer to monitor the validity of the bindings established to remote resources and to re-establish them when invalidated. The two repository components are responsible for encapsulating information about components already deployed locally (*LocalRepository* component), or available for download and deployment on remote hosts (*RemoteRepository* component). These components are thus responsible for evaluating the conditions set in the `<ADAPT>` elements of the application profile.

The Decision Layer. This layer encapsulates the evaluation and selection aspects of the Q-CAD protocol. After the Information layer has performed its pruning, the Decision layer evaluates the utility function against the shortlisted resource descriptors, and selects the one that maximises the application's utility (ties are broken randomly). As previously mentioned, the utility function can be either evaluated remotely, on the host that is offering the resource, or locally, provided that the remote resource sends its descriptor. The *Evaluation* component of the Decision layer thus comprises both a *Local* and a *Remote* component, for local and remote evaluation of the utility function respectively. More precisely, the Local component interacts with the Information layer to get the resource descriptors against which to evaluate the utility function locally. As we will discuss below, the Remote component uses the functionality provided by the Action layer instead, to distribute the utility function to the hosts on which it is going to be evaluated. The execution of the Evaluation component may generate events that need application input (see Section 3.3). If that is the case, the NotificationService component in the Application Meta-Interface layer is used to pass the events to the application and get the required input. Note that realisations of the Q-CAD architecture may combine the discovery message and the utility function in a single message. In this case, the Information layer and the Decision layer would work jointly to perform the matching and evaluation parts of the discovery protocol in a single step.

The Action Layer. This layer encapsulates the logical mobility techniques [12] required by the Decision layer (i.e., code-on-demand and remote evaluation). It consists of the *Deployer* component, which comprises: the *RemoteEvaluation* component, used by the Remote component in the Decision layer to deploy the utility function on a remote host, and the *CodeOnDemand* component, that is responsible for downloading any remote component locally needed to perform adaptation (tag `<ADAPT>` in the application profile). The downloaded components are registered with the LocalRepository, so that the Information layer maintains an up to date status of the system.

5 Q-CAD Implementation and Evaluation

The Q-CAD architecture has been implemented using Java 2 Micro Edition (Connected Device Configuration, Personal Profile) [21]. More specifically, we used the

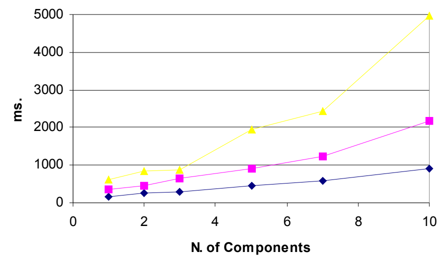

Figure 9. Impact of Application Profiles. Three cases are considered (from top to bottom): 5 contexts with 10 resources each, 3 contexts with 5 resources each, and 1 context with 1 resource each.

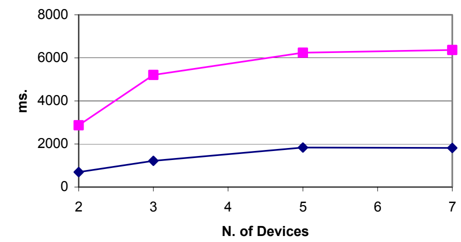

Figure 10. Overall Q-CAD Protocol Performance. Two cases are considered (from top to bottom): 5 components with 3 contexts and 5 resources each, and 3 components with 1 context and 1 resource each.

SATIN [24] component model and middleware system for adaptive mobile systems. SATIN provided us with an implementation of the Deployer component in the Action layer, as well as the LocalRepository, RemoteRepository and Discovery components in the Information layer. We used realisations of the Information layer employing both a multicast and a centralised publish-subscribe system to advertise and discover remotely available resources. The functionality provided by the SATIN middleware system occupies 89KB as a Jar archive. On top of that, the implementations of the application meta-interface and decision layers occupy 14KB as a Jar archive. We use the KXML2 [1] parser to access the application profiles and utility functions, which occupies a further 24KB as a Jar archive. In total, the Q-CAD implementation occupies 127KB (compressed), making it suitable for mobile devices.

We have implemented a benchmark application to evaluate Q-CAD performance in terms of: overhead imposed by the evaluation of context information (as encoded in application profiles), and overhead imposed by the evaluation of QoS information (as encoded in utility functions). All tests were performed with laptops with 128MB RAM and 300MHz i686 processors, which were connected in an ad-hoc network using 802.11b wireless (Wi-Fi) cards. The charts in this section display the average elapsed time over 20 resource discovery requests.

Figure 9 illustrates the impact of context-awareness, that is, the time taken by a mobile device to evaluate application profiles of various complexities. In particular, we have varied the number of alternative components (tag <ADAPT_COMPONENT>) in a range $[1..10]$; each component was associated with a different number of contexts (i.e., $[1..5]$), and each context was associated with a different number of local resources (i.e., $[1..10]$). In our experience, application profiles with 5 alternative components, associated with 3 contexts of 5 resources each, already represented by far the maximum level of context-awareness we needed (i.e., worst-case scenario). In this case, as the chart shows, the average amount of time to evaluate context is below 1 second. We have also evaluated the impact of QoS-awareness, that is, the time taken by a resource provider to evaluate the utility function that the querying agent distributes. Regardless of the number of parameters listed in the function, the elapsed time is constantly below 200ms.

Figure 10 illustrates the overall performance of the Q-CAD discovery protocol. We have varied the number of devices involved in the selection process in a range $[2..7]$; we have then measured the time elapsed from the moment a discovery process is started on a client machine, to the time a binding to one of these devices is established. This includes: the time taken to evaluate the application profile on the requesting host, the time needed to send around the query message inclusive of the utility function, the time taken to evaluate the utility function on each remote device, and the time needed to obtain answers and make a final decision. As the chart shows, for plausible profile configurations, the average amount of time is below 2 seconds, and it reaches 6 seconds for what we consider profiles of maximum complexity[2]. Also, the elapsed time does not sensibly increase with the number of devices involved in the selection.

These results demonstrate that the Q-CAD model and architecture are suitable for pervasive environments, as the overhead imposed on the devices is fairly small. An important lesson learned is that the most resource consuming task is, by far, the evaluation of context. In developing pervasive computing applications, context descriptions should thus be as simple as possible, containing only the needed information to discriminate between different adaptation components; utility functions are much less resource consuming, and thus provide a powerful mechanism to further refine the selection performed by application profiles.

[2]The results shown here do not consider peer failure; if peer failures are taken into account, the elapsed time depends also on the timeout values used before realising a peer is no longer within reach.

We have also evaluated the Q-CAD model and architecture in terms of usability and effectiveness of the abstractions it offers to application designers. In particular, we assigned an MSc student the task of implementing a prototypical tour guide application. The application is composed of an interface component, that can display information using 2 different viewer components: a component that is able to read and display plain ASCII text and one that can parse and display basic html (text and images) from a sensor. She reported that developing the tour application on top of Q-CAD required minimum effort: in particular, using the abstractions that Q-CAD provides (i.e., application profiles and utility functions) was straightforward, while the tedious task of querying heterogeneous sensors to gather and maintain context information was completely transparent. The most difficult task she was faced with was to gather end-user preferences and to design a synthesising algorithm that mapped user's preferences into application profiles. We believe these issues are not intrinsic to our model, but apply, in general, to scenarios where adaptation to changing contexts and user requirements is needed. Further research in this direction that involves both middleware, as well as HCI and requirement elicitation experts, is needed.

6 Related Work

In recent years, research has been very active in the area of service discovery for pervasive systems. Most of the work has concentrated on designing protocols and architectures that could fit the mobile network topology, characterised by frequent disconnections and changes.

Directory-based approaches, either centralised or distributed, have been very popular in both traditional distributed systems and nomadic scenarios. Examples include Sun Microsystem's Jini [4], Microsoft's Universal Plug and Play (UPnP [2]), the Service Location Protocol [13], the Salutation Architecture [19], and the Bluetooth Service Discovery Protocol [20]. Their applicability is, however, limited in highly dynamic mobile ad-hoc networks, where the existence of a service repository may not be assumed. Totally decentralised approaches based on flooding algorithms have been suggested (e.g., the Simple Service Discovery Protocol SSDP [11]); however, their heavy consumption of bandwidth and energy limits their applicability on portable devices. IBM DEAPspace [18] proposes a discovery protocol for single-hop ad-hoc networks, where each node caches service information locally; knowledge is built by broadcasting the local cache to neighbors, while service lookup is accomplished by searching the local cache. In mobile, multi-hop networks, Lanes [15] proposes protocols to efficiently manage the storage of service descriptions and the routing of queries, based on their semantics; RUBI [14] describes an efficient resource discovery framework that exploits routing algorithms to enable adaptation of the way information is disseminated and retrieved, based on a local view of the structure of the network. In peer-to-peer systems, the JXTA-Search [23] mechanism suggests a combination of broadcast and rendez-vous protocols to enable efficient discovery both within a small group and in a wider community. A common limitation of these approaches is that they concentrate on providing a communication infrastructure, while supporting only primitive service matching mechanisms, based on the exact match of simple predefined attributes, or leaving the format for information description and matching undefined. In pervasive environments, a generally-drawn service request may return a very large number of candidate services, and it may require an unbearable computation and/or cognitive load to select among them. Sophisticated models and protocols have to be designed, on top of this communication infrastructure, that enable context and QoS aware service description and matching. Q-CAD advances this goal by means of application profiles, utility functions, and a protocol that performs context and QoS aware resource matching.

A technique that moves a step closer to our goal is semantic routing. It uses the flooding algorithm as a basis; however, rather than forwarding a query message to every neighbour node, each node intelligently chooses a subset of them based on the semantics of the request. This is usually achieved by means of an application-layer overlay, as in the Intentional Naming Scheme (INS) [3] and in the agent-based discovery framework Allia [10]. While moving a step in the right direction, these approaches are only able to handle simple descriptions; Q-CAD, instead, supports richer and more complex descriptions and queries, while keeping the computational overhead low.

MAGNET [16] is a trading framework that has been proposed to allow user-customised service matching. Discovery is based on service types, rather than on service names; the criteria used to select the best service provider (e.g., the nearest) can be customised. Although interesting, this approach is limited: its tuple-space based matching mechanism, in fact, does not allow semantically rich inexact matching. In [17], a QoS-aware service selection framework for mobile ad-hoc networks is described, that takes both the user perspective and resource consumption into account. However, the expressiveness of QoS issues is poor: for example, only a small, fixed number of resources are considered; moreover, users can only prioritise among them, without the possibility of binding these preferences to context conditions. Q-CAD builds on top of our previous research in the area of context-awareness [6] and component-based adaptation [24] to provide a unique resource discovery model that enables pervasive computing applications to efficiently discover and select the resources that can best satisfy their user, based on semantically rich descriptions of both the current context and QoS needs.

7 Conclusions

This paper has described Q-CAD, a QoS and context aware resource discovery framework for pervasive environments. Q-CAD combines expressive specifications of user preferences with efficient processing. In particular, users specify the context conditions that should influence the discovery and selection of resources in application profiles, while the non-functional requirements are encoded in utility functions. Q-CAD discovery and selection protocol efficiently uses the information contained in the profiles to prune the number of matches; it then uses utility functions to select the best resource out of the pruned ones. As shown in the Q-CAD evaluation section, application profiles and utility functions allow semantically rich queries and matching, imposing a low overhead on the device.

Our plans for the future point to three directions. First, in this paper we concentrated on evaluating Q-CAD performance; however, we realise the importance of assessing Q-CAD usability as well. In order to do so, we plan to develop a broader number of applications on top of our architecture, and thus gather enough experience to provide application developers with guidelines on how to use application profiles and utility functions best. Second, we plan to remove the assumption that a universally accepted ontology is being used, as we believe that such an ontology will never exist. Rather, we intend to extend the Q-CAD framework so to be able to dynamically perform semantic translations between different ontologies. These translations will not be exact: a new concept of 'probable' match will thus have to replace that of an 'exact' match. Finally, in this paper, we made the assumption that all the entities involved in the discovery process were trustworthy; it is our plan to extend the Q-CAD framework with a trust management model we have developed [5] and that enables mobile devices to dynamically assess the trustworthiness of the other entities they deal with, based on their past interactions and recommendations sent by other entities in the pervasive setting.

References

[1] The KXML2 XML parser. http://kxml.enhydra.org/.

[2] Universal Plug and Play. http://www.upnp.org/, 1998.

[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201. ACM Press, 1999.

[4] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[5] L. Capra. Engineering human trust in mobile system collaborations. In *Proc. of the 12th Int. Symposium on the Foundations of Software Engineering*, pages 107–116, Newport Beach, CA, November 2004.

[6] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, Oct. 2003.

[7] L. Capra, S. Zachariadis, and C. Mascolo. Q-CAD Specification Language. http://www.cs.ucl.ac.uk/staff/L.Capra/projects/qcad/.

[8] H. Cervantes and R. Hall. Beanome: A component model for the OSGi framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne, September 2002.

[9] H. Cervantes and R. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proc. of the 26th Int. Conf. of Software Engineering*, pages 614–623. ACM Press, May 2004.

[10] O. R. et. al. Allia: Alliance-based Service Discovery for Ad-Hoc Environments. In *Proc. of ACM Mobile Commerce Workshop*, Sept. 2002.

[11] I. E. T. Force. Simple Service Discovery Protocol, Oct. 1999.

[12] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[13] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location protocol, Version 2. Technical Report RFC 2608, IETF, June 1999.

[14] R. Harbird, S. Hailes, and C. Mascolo. Adaptive Resource Discovery for Ubiquitous Computing. In *Proc. of the* 2^{nd} *Int. Workshop on Middleware for Pervasive and Ad-Hoc Computing*, pages 155–160. ACM Press, Oct. 2004.

[15] M. Klein, B. Konig-Ries, and P. Obreiter. Lanes - A Lightweight Overlay for Service Discovery in Mobile Ad Hoc Networks. In *Proc. of the 3rd IEEE Workshop on Applications and Services in Wireless Networks (ASWN2003)*, Berne, Switzerland, July 2003.

[16] P. Kostkova and J. McCann. Support for dynamic trading and runtime adaptability in mobile environments. pages 229–260, 2003.

[17] J. Liu and V. Issarny. QoS-aware Service Location in Mobile Ad Hoc Networks. In *Proc. of the 5th IEEE International Conference on Mobile Data Management*, Berkeley, USA, Jan. 2004.

[18] M. .Nidd. Service Discovery in DEAPspace. *IEEE Presonal Communications*, pages 39–45, Aug. 2001.

[19] Salutation Consortium. Salutation. http://www.salutation.org/, 1999.

[20] B. SIG. Specification of the Bluetooth System, Feb. 2001.

[21] Sun Microsystems, Inc. Java 2 Platform, Micro Edition. http://java.sun.com/j2me/, 2000.

[22] Urban tapestries. http://urbantapestries.net/.

[23] S. Waterhouse. JXTA Search: Distributed Search for Distributed Networks. http://search.jxta.org/.

[24] S. Zachariadis, C. Mascolo, and W. Emmerich. SATIN: A component model for mobile self-organisation. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 3291, pages 1303–1321, Agia Napa, Cyprus, October 2004. LNCS, Springer.