Context-Aware Service Selection Based on Dynamic and Static Service Attributes

Steve Cuddy, Michael Katchabaw, and Hanan Lutfiyya

Abstract -- **Context-aware applications are able to use context, which refers to information about the surrounding environment, to provide relevant information and/or services to the user. A context-aware application may need to make use of existing services (e.g., a print service). There may be several possible choices of services. The context–aware application should be able to discover and select a service that considers context (e.g., current user location). Existing architectures and protocols for service discovery, however, are not suitable for doing so. Contextual information, by its very nature, is dynamic, reflecting the current state and conditions of the application, its user, or its operating environment. Existing architectures and protocols for service discovery, however, tend to assume the world is static, with attributes describing services offered never changing. If attributes are allowed to change, the approaches do not provide the architectural mechanisms required to update them; dynamic attributes with no means of updating are static for all intents and purposes. To support context-aware service discovery and selection, a better approach is required. This paper discusses one possible approach that is based on existing techniques.**

Index Terms: **Service discovery, monitoring, service selection, context-aware applications.**

I. Introduction

The increase in usage of wireless networks and mobile devices implies that devices and services are entering and leaving the network with increased frequency. This results in a more dynamic computing environment and increases the need to allow a client device or service (hence referred to as *client*) to discover other devices and services in the network. This is referred to as *service discovery*. Devices are hardware that typically provide a service. For example, a printer is a hardware device that provides a print service. In this case, a printer is both a device and a service. Not all services are devices, however. For example, a service may be software running on a server that has many other services running on it e.g., address book. In this paper, the terms device and service are used interchangeably. As can be seen from this discussion, the definition of a service is not rigid. Essentially a service is a facility or software available to users or to applications. Examples of service discovery protocols (SDPs[1]) include Bluetooth [1], Jini [22], Universal Plug and Play (UPnP) [25], Service Location Protocol (SLP)[6], and Salutation [14].

Consider the following example illustrating the usefulness of service discovery protocols. The user is at an unfamiliar location. Assume there is an SDP environment at this location that allows for discovery of a printer. The client is able to request a printer using the SDP environment and receive information on one or more available printers. The advantage of the SDP environment is that the client does not need to know the name of the printer, the IP address, and in some SDPs, does not need to install drivers, knowledge on how to install it using a wizard, etc. This is handled by the SDP.

A *service type* refers to a category of services or devices. A service type specifies attributes that are used to characterize the category of services or devices that the service type is associated with. Examples of attributes for a printer service type include pages per time unit, mode (duplex or single), and an attribute representing if color is supported. SDPs provide facilities for finding services by specifying conditions on attribute values or providing facilities to allow a client application to ask a device for information on attribute values.

SDPs essentially assume that the attributes used to characterize a service type are static. The values of static attributes are assigned when the service is first deployed. This limits the support of *context-aware* applications. A *context-aware* application uses information about the circumstances that the application is running in (i.e, context), to provide relevant information and/or services to the user. In the printer example, several printers satisfying the user's requirements of the printer being laser, color and duplex may be discovered. The user may also want a printer that has the fewest print jobs and is on the same floor as the user. Since a user's location changes, the printers on the same floor as the user may also change. In addition, the number of print jobs at any printer dynamically changes. The set of printers on the same floor as a user and the number of print jobs associated with a printer

This work was supported in part by the National Science and Engineering Research Council of Canada (NSERC).

S. Cuddy recently completed his MSc degree at the University of Western Ontario

H.Lutfiyya is with the Department of Computer Science at the University of Western Ontario, London, Ontario Canada N6H 5C9 (phone: 519-661-2111; fax: 519-661-3515; e-mail:hanan@csd.uwo.ca).

M. Katchabaw is with the Department of Computer Science at the University of Western Ontario, London, Ontario Canada N6H 5C9 (phone: 519-661-2111; fax: 519-661-3515; e-mail:katchab@csd.uwo.ca).

[1] Throughout this paper, the term SDP refers to a generic service discovery protocol. It should not be confused with the Bluetooth SDP.

are represented by *dynamic* attributes. The value of a dynamic attribute changes after the service has been deployed. Since existing SDPs assume that characterization of services is based on static attributes, the existing SDPs do not provide adequate support for context-aware applications.

Although SDPs allow the definition of attributes that are dynamic, none of the SDPs provide facilities to monitor the dynamic attributes (hence the reason why SDPs are considered to support only static attributes). This work addresses this by developing an approach that allows for the use of dynamic attributes in the presence of existing SDPs. This allows for the discovery and return of a service to a requesting application based on contextual information as defined by dynamic attributes. The use of existing SDPs allows current deployments of SDPs to remain in place.

This work considers that the importance of a specific dynamic attribute varies among users. In the printer example, some users will prefer to use the printer with the fewest number of jobs waiting to be processed while other users will prefer to use the nearest printer. A user's preference is not necessarily static.

This paper presents an architecture and implementation of this architecture that provides services to support service selection based on information that is not necessarily static and allows user preferences to be taken into account. An important aspect of this work is to be able to use, as much as possible, existing monitoring tools and SDPs. The paper is organized as follows: Section II briefly describes existing SDPs, Section III describes related work on service selection, Section IV describes the architecture, Section V describes the prototype implementation and its application to two service types, Section VI provides a discussion and Section VII provides a conclusion and future work.

II. Brief Description of Service Discovery Protocols

The need to be able to discover devices has driven the development of service discovery protocols. These are briefly described in this section.

The Service Location Protocol (SLP) was developed by an IETF working group [3] with the intention that it could be used for large enterprise networks that use TCP/IP. There are three different entities that can be present in an SLP environment. These are the user agent (UA), service agent (SA), and directory agent (DA). The DA is optional. The UA initiates service discovery on behalf of a client. A query for a specific service type is sent to SAs through multicasting or to a DA via unicast. An SA is associated with a service or device that is advertising itself to be discovered. A DA is a centralized information repository that makes itself known by multicasting a message about its presence. A DA accepts service registrations from an SA and responds to UA queries. When an SA registers, it provides values of attributes associated with its service type. When a UA sends a request to a DA, the DA checks its database for matching entries and returns a URL for each service found. DAs are not used in smaller SLP environments. The use of a DA is for scalability. This allows the UA to find services by sending its request to the DA directly via unicast, otherwise the UA multicasts its request on the network so that all available SAs can receive the request. Requests sent to the DA or SAs can be matched on the required attribute values requested by the UA. These attribute values can be combined into Boolean expressions using AND operators, OR operators, common comparators {=, >, <, >=, <=} and substring matching.

Universal Plug and Play (UPnP) [13],[17], developed by Microsoft, provides for service discovery in small to medium size networks. On a periodic basis, devices advertise themselves. Clients that need to discover a service run a control point that waits for advertisements from devices or the control point can actively multicast a message specifying the desired service type. Devices that receive this message respond by sending a unicast service advertisement. A client can retrieve an XML description of the device that includes the attribute values of the associated service. Thus, a client can filter out devices.

Sun Microsystems' Jini Technology [23] has a lookup service that stores information about the services available in the Jini community. It stores attribute descriptions for the services. Jini services must register with a lookup service on the network if they wish to join the community. When a client needs a service, it sends a query to the lookup service. Searching based on specific values of attributes is possible. If a client requires additional software to be installed to make use of the service, Jini uses RMI. This allows the additional software, typically device drivers, to be downloaded via object code from the service and then, in turn, be executed on the client. Jini requires that each client has a JVM.

Salutation has an entity called the Salutation Manager (SLM) [14],[18],[19]. An SLM provides a service similar to the lookup directory of Jini or the DA in SLP. Although the SLM does not store service attribute data, attribute-based searching is possible in a very limited sense by allowing the client application to query the service directly. For example, when searching for a printer with capabilities of printing in color, a minimum of 10 pages per minute, and with duplex capability, the client retrieves a list of print services from the SLM. It can then query each of the returned print services to see if they have the desired values.

Bluetooth [1] is for ad-hoc, short-range, wireless networks. The Bluetooth wireless technology "is designed to replace cables between cell phones, laptops, and other computing and communication devices within a 10-meter range" [1]. For example, Bluetooth could be used to discover a local printer wirelessly. Bluetooth has the ability to search for specific

service types, and on a very limited basis, to search based on service attributes.

Web service architectures use service directories specified using UDDI [24] that are registries of service descriptions. Services discovery is based on static parameters.

Each of the SDPs can either filter based on conditions on attribute values or allow clients to filter using the facilities provided by the SDP to query services for more information. Although it is possible to define dynamic attributes, none of the SDPs provide mechanisms to monitor the values of these attributes, nor do they automatically choose a service [26].

III. Related Work

The previous section briefly described existing and commercially available service discovery protocols and concluded that none of these provide support for dynamic attributes. These service discovery protocols provide information about services either using a centralized directory that stores information about services or by having services broadcast information about their services. These approaches do not work in an ad-hoc network. There is work that is context-aware that is used to determine the services available in an ad-hoc network (e.g., [10]). The work in this paper is different in that the emphasis is on selecting a service from a set of services that satisfy the constraints placed on the static attributes considering context. The work in this paper can assume that services are discovered using techniques such as that described in [10]). There is work that uses context in a limited sense. For example, the work in [16] uses physical location. The work in this paper potentially can deal with any context and is based on an existing protocol.

The closest related work is presented in [8]. This work discusses an approach to choosing the service best service based on a weighting system. A Service Discovery Model (SDM) was created to allow for comparison of the service discovery protocols. Their approach requires a modification of the SLP environment. In SLP, a directory agent (DA) collects information advertised by services and stores that information in a repository. The work in [8] modifies the DA so that it ranks the services. Information about the service with the highest ranking is returned to the requestor. The work in this paper is different in that it does not modify existing SDPs. The advantage of this includes the ease of portability to other architectures. The work in [8] does not discuss the issues needed to maintain information that dynamically changes e.g., printer queue size.

The work in [12] also introduces the use of dynamic attributes to service discovery protocols. The paper describes the interface that an object representing a dynamic attribute must present. This interface includes methods that return a value and the time that the value is considered valid to the requesting entity. This is incorporated into a three-tier service discovery architecture. The work described in this paper differs in that the use of existing SDPs and monitoring facilities is assumed.

Another closely related work is described in [5]. This work provides facilities to allow for user annotation of services and usage history. However, this work developed a new system to do so. This work presented in this paper is able to interact with existing systems.

The Cooltown project [2] is able to take context into account, but in a limited sense. For example, a hotel may have two printers in the same wireless network, but each of the printers is in a different room. Policies are needed to determine which printer a guest may actually have access to. Thus, Cooltown can take context into account by considering the location of the user. It does not take into account the context associated with a service. There are also examples of work e.g., [4] that describe a model for context information and efficient searching of context information.

IV. Architecture

This section describes the architectural design. This discussion focuses on components and their interactions. It is assumed that the installation of additional components (e.g., a print service requires printer drivers to work properly) is handled by the SDP or some other mechanism. The goals of the architecture design were the following: (i) Existing SDPs should not be changed; (ii) The design should not be limited to a particular service; (iii) The design of the architecture should be able to allow selection based on attributes whose values dynamically change e.g., the size of the print queue or the load of the machine that a service may be located on; (iv) The architecture should allow for service selection to be as automated as desired by the user which may mean that it is completely automated without user intervention; (v) Different users should be able to place different emphasis on criteria used in selecting a service; (vi) The interface to the user should be friendly, simple and elegant.

A. Dynamic Service Attributes

The *dynamic service attributes* are those characteristics of a service whose values change over time. Dynamic service attributes are used to characterize context. Otherwise the attribute is said to be *static.* An example of a static attribute for a printer is the number of pages that it can print per minute. An example of a dynamic attribute for a printer is the number of prints jobs in the print queue. This is essentially a measurement of the load of the printer. Although SDPs allow for attributes that represent a dynamic attribute, the SDPs do not handle the monitoring of these attributes. Thus, this work assumes that only static attributes are to be defined within SDPs. The architectural components associated with the service selection will focus on dynamic attributes. Dynamic attributes used to select a service can be associated with the

service to be discovered or with the client. Dynamic attributes can be calculated from other attributes. For example, a client may be specifically interested in the nearest service. The nearest service depends on the location of the client (which changes if the client is mobile) and the location of the service. A dynamic attribute representing the distance from the client's location to the service location is calculated from the client's and the service's current locations.

B. Weight Vectors

The importance of a dynamic attribute for a specific service may differ for different clients. For example, one client may place a high importance on speed, while another client may place a higher importance on the location of the service. Assume that for a specific service type i and a client that is identified by j that $W_{ij} = (w_{ij1}, w_{ij2}, \ldots w_{ijn})$, where $w_{ij1} + w_{ij2} + \ldots + w_{ijn} = 1$ is the set of weights associated with service type i for client j. The weight w_{ijk} is the weight that client j assigns to service i for the kth attribute. In the printer example, assume that the dynamic attributes are the size of the print queue and the distance between the client and the service. For client j, $W_{printer,,j} = (0.50, 0.50)$. There is a weight of 0.50 associated with the print queue and 0.50 is associated with the distance. This suggests that equal priority is given to the two attributes. The weight vector $W_{printer,,j} = (0.90, 0.10)$ suggests that a much higher priority is to be placed on the size of the print queue. The next section will describe how weight vectors are used.

C. Architectural Components

The architecture (in Figure 1) shows a service discovery protocol environment that includes the enhancements provided by service selection. The boxes in Figure 1 represent service entities and the arrows show data flow.

D. Client Device

The client can be any device looking for a service on the network. It is a consumer of services provided by others. Functionality is required to allow participation in the service discovery protocol environment. The client requires components that encapsulate this additional functionality. The client must be able to initiate the discovery process of services in a service discovery protocol (SDP) environment. This is encapsulated in the *Service Discovery Enabling Component*. When the client requires the use of a new service, it makes a request for that specific service. If the client receives more than one service in response to its request, it initiates the service selection process. This is encapsulated as the *Service Selection Enabling Component*. The service selection process finds a Management Console Service (MCS) on the network. As an option, the client can send its weight vector for the service type to MCS. If there is no weight vector, a default vector is used. The MCS selects a service based on the weight vector and returns this to the client.

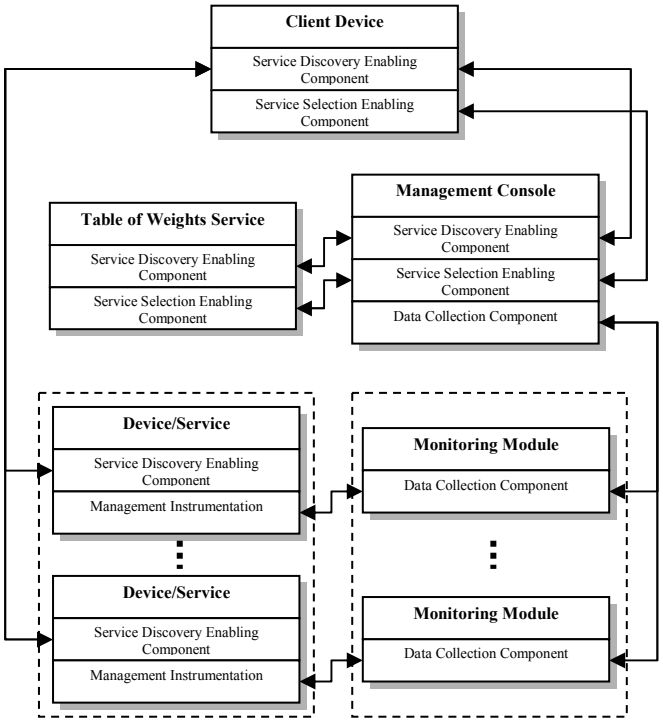

Figure 1. Architectural Components

The separation of the *Service Discovery Enabling Component* and the *Service Selection Enabling Component* allows for different SDPs to be used. The client application initiates service discovery as before. If the client is expected to do its own filtering based on the values of the static attributes (as must be done in UPnP), it does so before sending services to the *Service Selection Enabling Component.*

E. Management Console Service

The Management Console Service handles the decision making process of service selection for the client. The *Service Discovery Enabling Component* is used to advertise the availability of the MCS on the network. It is a service that is to be discovered by the client. It is assumed that this component can be implemented to support multiple SDPs. Once the client has established communication, it receives a list of services from the client and optionally the client's weight vector. If the client does not send a weight vector, the MCS uses a weight vector that it retrieves from the Table of Weights (TOW) Service. The MCS discovers this service using a SDP. This discovery is encapsulated in the *Service Discovery Enabling Component*.

The MCS makes a selection decision based, among other things, on the values of the dynamic service attributes. Values of dynamic attribute information are collected from monitoring tools. This is handled by the *Data Collection Component.* The decision making process is encapsulated in the *Service Selection Enabling Component.*

Based on the values for dynamic service attributes gathered, a weight vector, and policies, a service can be selected from

the multiple devices and sent back to the client. This component first computes a *ranking* of services. Ranking is based on the computation of a score for a service. For each service type *i* and client identified by *j,* the score is computed as follows: $\Sigma DSA_{ijk} * w_{ijk} * f_{ijk}$. where DSA_{ijk} represents the monitored value that the Data Collection Component returned for attribute DSA_k when client *j* needed that information for service type *i*, w_{ijk} is the weight that client *j* assigns to service type *i f*or the *j*th attribute. The value of f_{ijk} is either -1 or 1. This means that the product of $DSA_{ijk} * w_{ijk}$ can be added or subtracted from the total score.

The service with the lowest score is not necessarily the service that should be sent back to the requesting client device. Consider that the scores assigned to printers P_1,P_2,P_3 results in this ranking of the printers: P_2,P_1,P_3 . The administrator may have a limit on the number of print jobs allowed. If the addition of the print job from the requesting client would exceed this limit, then P_1 should be selected. This is regulated by policies [21]. A policy is a rule that is generally of the form of an *if-then* statement. These rules are not hard-coded in the MCS. Rather, they may be retrieved from a management agent. This allows for the MCS to be adaptable.

F. Table of Weights Service (TOW)

This component maintains a default weight vector for each service type. There are two components that make up the TOW service. As with any service in a SDP environment, there is the component that allows its participation in this environment. This is the *Service Discovery Enabling Component*. The *Service Selection Component* receives a request for a default weight vector for a specific service type, e.g., print service, from the Management Console Service. Once the TOW service has determined the correct weight vector, it sends this vector to the MCS.

G. Monitoring Modules

For each of the services listed in the table of weights service, the ability to dynamically monitor the services must be present. This is required so that once the weights for each of the dynamic service attributes have been received from the TOW service, the actual values for those dynamic service attributes can be gathered, and then used in service selection.

A monitoring module is specific to a service type. When a service type is added, it is registered with the MCS. The registration includes a list of attributes that are monitored and a *handle* that is used by the MCS to communicate with a process that belongs to the monitoring module that coordinates the monitoring. For example, a set of printers may be monitored using SNMP. A process is contacted by the MCS that then can use SNMP to collect information from a printer, yet also make use of existing monitoring approaches and tools. This design allows for the use of a standard interface to the process that coordinates the monitoring. The MCS sends the names of the dynamic attributes and service instantiation that it is interested in, and the coordinator returns the value of the dynamic attributes. The coordinating process hides whether or not the monitored dynamic attributes are monitored using a push or pull mechanism and the monitoring protocols.

H. Interactions

This section describes the general selection process when there are multiple services discovered which meet the static service selection attributes. The general process is briefly described as follows:

- An application on the client device determines the need for a specific service with values for specific attributes. A request is made to an SDP.
- The SDP returns a set of services that satisfy the request. If necessary, the application on the client device filters the services based on values of the static attributes. This is necessary for those SDP environments that do not provide facilities for filtering.
- If there are multiple services to choose from, the client application can either select a service randomly or initiate a service selection process, which uses more information, by discovering a Management Console Service. If multiple MCSs are discovered, the client application randomly selects a MCS.
- The client sends the MCS service the service type, a list of discovered services that it requires service selection to be performed on, and perhaps a default weight vector.
- If the MCS does not receive a default weight vector then it requests one from the TOW service. The MCS discovers a TOW service. If multiple TOWs are discovered, then MCS randomly selects a TOW service.
- The MCS requests monitored information for the dynamic service attributes.
- The MCS now has all the information required to perform service selection on the services as requested by the client. Using this information, the MCS determines the best service and returns this to the client device's application.

V. Prototype and Experiments

This section discusses the implementation of the architecture discussed in Section IV.

SLP was chosen as the service discovery protocol to use for the prototype environment based on these factors: (i) SLP is considered the most mature of the SDP technologies; (ii) There is an open-source implementation available which is well documented; and (iii) There are numerous examples of SLP code that exist. The open-source implementation of SLP selected for use in the prototype was OpenSLP, developed by OpenSLP.org, Caldera Systems Inc., and independent contributors via SourceForge [15].

The experimental environment in this work is our research lab, consisting of a variety of Solaris, Linux, and Microsoft Windows workstations. The various elements of our architecture, as well as the requisite SLP components were deployed throughout this environment.

A. MCS and TOW Implementation

The MCS and TOW are written in C, compiled and executed on a Solaris system. The initialization process of SLP requires that it register with an SA. Both wait for incoming client connections. Each listens on a TCP/IP port specified via the command line when executing the MCS.

B. Services Implemented

Two service types were considered. One is used to represent printers and the other is used to represent workstations. These services were chosen since they have dynamic attributes associated with them, there are already existing monitoring tools for many of these dynamic attributes and the potential usefulness of using these service types (Students often comment on how useful it would be to know where the nearest free workstations are). Both of these service types have been used in context-aware applications. The purpose of using two services is to illustrate that the architecture is not limited to a specific service or a specific monitoring approach.

1) Print Service

The print service implemented for this work is a simulated print service. There are two parts to each simulated print service. The SLP-enabling component allows the print service to respond to the multicast requests from the client. The print service initializes by registering itself with a Service Agent. The second component allows the print service to respond to queries about dynamic attribute information. The simulated print service was deployed on multiple Solaris workstations during experimentation.

The monitoring of these dynamic attributes was done through SNMP. SNMP (Simple Network Management Protocol) is a popular network management protocol. The product selected for use in the development of the prototype was Net-SNMP v5.1.1 [11] NET-SNMP is an open source implementation of the SNMP protocol. Many printers, network devices and host machines come with an SNMP agent. To enable the print services to work with SNMP, we used a greatly simplified version of the standard printer SNMP MIB (Management Information Base[2]). The dynamic attributes whose values are monitored using SNMP are the following: print queue length, toner remaining, and paper remaining. SNMP *get* queries are made only at the time the MCS needs to calculate which service to select based on a client's request – there is no polling of the services. A location process provides information on the location of a user. This information is retrieved from a table. Future work will have this process collect location information from a location system.

2) Workstation Service

The purpose of the workstation service is to locate workstations that are available for use. A small monitoring agent is deployed on each Solaris, Linux, and Windows workstation to record its availability and number of users to a web service, composed of an Apache web server and MySQL database. A separate SLP proxy periodically queries the web service using HTTP and SQL requests to determine which workstations are available, as well as their static attributes (including their name, address, operating system, and location), and registers this information with a Service Agent. One registration is made for each workstation found so that multiple workstation services can be located with each SLP query.

Dynamic attribute information is collected on demand by the MCS through additional HTTP and SQL queries to the web service tracking the workstations. The attribute in question here is currently only the number of users on the workstation, but this can be easily expanded in the future to include a variety of resource utilization metrics.

C. Client Implementation

The client in the prototype was built with the purpose of simulating an application that is SLP-enabled and that requires the use of a print or workstation services. The client is on one of the Solaris systems. The client implementation is platform independent, with versions available for multiple flavors of Unix (Solaris and Linux), with ports in progress to Windows, PocketPC, and other portable environments. Unix was chosen as the initial client platform because of the ability to quickly develop and test a prototype in this environment. The software functionality does not change for different versions created for different operating systems.

D. Experimentation

Experimentation consisted of running the test client multiple times to generate service discovery requests for our provided services with various weight vectors and attribute filters. In all cases, the selected service was chosen appropriately based on the parameters provided by the client. This verified that our prototype system behaved correctly.

Experiments showed that service selection took almost as long as service discovery. The dominant cost in service selection is discovering MCS and TOW services. Section VI discusses changes in the architecture that can reduce this cost.

[2] A MIB is a data file containing a collection of all objects managed in a network. Objects are variables containing the state of processes running on a device. They may also contain text information about the device, such as a name and description.

VI. DISCUSSION

This section briefly discusses the observations of the architecture and working with the prototype.

Improved Support for Context-Aware Applications. This prototype extends support for context-aware application by handling context that is defined by dynamic attributes: user location information and dynamic information that characterizes load of printers and workstations. Although some of this information was emulated (e.g., location) we were able to illustrate the feasibility of the proposed architecture. We showed that a coordinating process that collects monitored information can be used to hide from the MCS the details on how the information is collected (e.g., is it being collected using SNMP or a location system). We showed that this information can be used to select a service.

Architecture Variations. The work presented in this paper showed the SDPs returning services to the client application. The client application could be one that is executing on a resource-constrained mobile unit that conceivably could be overwhelmed with the number of services returned. The architecture easily allows for a proxy for the client to be used that is executing on a more powerful machine.

As discussed in Section V, experimental results show that the service selection takes almost as much time as the initial service discovery. There are variations on the architecture that can be used to reduce the amount of time that service selection takes. Section V suggests that the MCS be discovered by the client and that the TOW be discovered by the MCS using an SDP. SDPs return a communications handle for further communication. The MCS can cache the handle for the TOW. If the handle becomes invalid, then the MCS can use an SDP to discover a new TOW service. If it is assumed that the MCS and TOW do not often change their locations, then this would reduce the amount of time for service discovery. It is also possible for the client to cache the handle of the MCS. The client would cache a handle for a specific environment e.g., there may be a handle cached for each organization. There is most likely limited space for storing handles and thus the client would have a limited number of entries. A least recently used algorithm can be used to replace entries with newer entries.

Currently the architecture has a set of services returned to the client from a SDP. The client then sends the returned services to MCS for selection based on dynamic attributes. The reason for using this approach is that it made it easier to use an existing SDP. An alternative approach would be for the client to send a service discovery request to the MCS. The MCS would use an existing SDP to find a set of services based on static attributes. It would then proceed to select from this set of services based on dynamic attributes.

Another architectural variation is that the functionality implemented by the MCS is moved to the client device. This would mean that the client would contact the monitoring services to get values of dynamic attributes. The disadvantage of this approach is that the client device would have to either discover the monitoring services (which does have overhead) or already know where to retrieve the monitored values of the dynamic attributes. The other disadvantage is that it would not be possible to support administrator policies (this was discussed in Section IV.E).

Reliability. Two new services were added: MCS and TOW. It is possible to replicate these services and thus since these can be discovered by an existing SDP a certain amount of fault tolerance can be provided.

Use of Multiple SDPs. The architecture supports the simultaneous use of different SDPs. This is possible by implementing the MCS and TOW services so that the *Service Discovery Enabling Component* has code needed for the different service discovery protocols. Thus the same MCS can be used for different clients using different SDPs. It should also be relatively easy to add or change the SDPs being used by the MCS and TOW services without having to change the software for actual selection.

Adaptability. The architecture is designed for adaptability. For example, although discussion focused on returning a single service to the client application, the architecture allows for dynamic attributes to be returned for each service. This would allow each service to be annotated with information. The client can then make the final selection decision. This sort of annotation has been shown to be useful in [5].

The architecture is not limited to using dynamic attributes when selecting a service. A combination of both static and dynamic attributes can be used with weights assigned to static as well as dynamic attributes. This allows for more flexibility in service selection.

Weight Vectors. Currently, the work assumes that different users will have different weight vectors to take into account their preferences. It should also be possible to consider that a user's preference changes over time or is different depending on the time of day e.g., sometimes proximity is more important than printer load and at other times the reverse is true. A user's preferences may depend on the domain that they are in. Future work should take this into account.

Adding New Service Types. A new service type is easily added by registering with the MCS. This registration includes names of dynamic attributes. The implication is that not all dynamic attributes have to be known at startup. This was validated by first providing a print service and then adding a service for finding free workstations. It should be possible to provide a web service [9] associated with each service type that provides operations for monitoring. This allows the MCS to assume a standard approach for requesting information without being concerned on how that data is actually collected. Our experience the printer and workstation types

showed that this was possible since for both these types the monitoring was done independently of this work. The workstation monitoring was done without knowledge of the work described in this paper.

Efficiencies in Monitoring. Currently, the architecture assumes the MCS requests values of dynamic attributes from the coordinating monitoring process for the requested specific type. The architecture can be modified to allow not only values of dynamic attributes to be returned but also the amount of time these values can be considered to be valid. This allows the MCS to cache these values for the valid time period, which results in fewer queries for monitored information.

Personalization. The architecture provides personalization by allowing a client to maintain information about a weight vector for each service type. It is possible to maintain information about a weight vector for each service type for each organization, a weight vector to be applied to all environments or no weight vector. Currently, the architecture maintains a weight vector to be used as the default for all requesting clients. It should be possible for the TOW or some other management service to keep track of selections made by clients. Thus, the next time the client requests a service of the same type it may be able to look at the history of selections and make a selection based on that history.

VII. Conclusions and Future Work

This paper presented an architecture for service selection based on the use of an existing SDP which selects based on static attributes and dynamic attributes, and taking into account preferences. This improves the suitability of using SDPs for context-aware applications. It was demonstrated that several management components could be added and used with an existing SDP. Initial results are promising since the overhead associated with service select does not dominate.

Future work includes the following: 1) Further study of architecture variations such as those described in Section VI with an emphasis on proxies. As part of this research, we will measure the performance impact that a proxy has when the client is a mobile unit. We will also examine the use of caching as an approach to reducing the time it takes to carry out service selection. 2) Expand the prototype to a larger environment. Currently, we are investigating using two campus buildings for the prototype. We are currently implementing a location system to be used in conjunction with this prototype. This location system becomes another monitoring module. This expansion will be used to enhance the user interface and provide more input into personalization. 3) This work currently assumes that the services to be discovered are not in a mobile ad-hoc network. The client device may be mobile but the services are not. Future work will examine the discovery of services in ad-hoc networks.

VIII. References

[1] Pravin Bhagwat. "Bluetooth: Technology for Short Range Wireless Apps", IEEE Internet Computing. Volume 5, No. 3, May/June 2001.

[2] Tim Kindberg et al, "People, Places, Things: Web Presence for the Read World", Technical Report HPL 2000-216 Internet and Mobile Systems Laboratory, HP Laboratories Palo Alto, February 2000.

[3] Caldera Systems Inc. "An Introduction to SLP", Whitepaper (Draft).

[4] Christos Doulkeridis, Efstratios Valavanis and Michalis Vazirgiannis. "Towards a Context-Aware Service Directory", Technologies for E-Services, Lectures Notes in Computer Science, volume 2819, September 2003, pp. 54-65.

[5] A. Friday, N. Davies, N. Wallbank, E. Catterall and S. Pink. "Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments", ACM Baltzer Wireless Networks (WINET) Special Issue, 10(6), 2004.

[6] E. Guttman, C. Perkins, J. Veizades, and M. Day. "Service Location Protocol, Version 2", Request for Comments (RFC) 2608, June 1999.

[7] C. Hertel. "Understanding the Network Neighborhood", Linux Magazine, May 2001.

[8] E. Hughes, D. McCormack, M. Barbeau and F. Bourdeleau. "Service Recommendation using SLP", IEEE International Conference on Telecommunications(ICT), Bucharest, June 2001.

[9] Kerry Jean, Alex Galis and Alvin Tan, "Context-Aware GRID Services: Issues and Approaches", First International Workshop on Active and Programmable Grids Architecture and Components, Lecture Notes in Computer Science 3038, 2004, pp. 166-173.

[10] M. Kheder and A. Karmouch, "Enhancing Service Discovery with Context Information", ITS'02, Brazil, September 2002.

[11] The NET-SNMP Home Page. Last access Sept. 2004. Available at http://www.net-snmp.org/.

[12] C. Lee and S. Helal. "A Multi-Tier Ubiquitous Service Discovery Protocol for Mobile Clients", First ACM Conference on Mobile Systems and Applications, May 2003.

[13] Microsoft Technet, "Universal Plug and Play in Windows XP." Technical Document, August 2001, available at http://www.microsoft.com/technet/prodtechnol/winxppro/evaluate/upnpxp.mspx.

[14] B. Miller and R. Pascoe, "Salutation Service Discovery in Pervasive Computing Environments", Technical Report, IBM White Paper, February 2000.

[15] OpenSLP web site. Last Accessed Sept. 2004. Available at http://www.openslp.org/#Credits.

[16] Peter Orbaek, "The OpenLSD Framework: Location-based Service Discovery", Department of Computer Science, University of Aarhus, 2003.

[17] Karen Reff (kreff@vtm-inc.com), UPnP(TM) Forum Administrator and Toby Nixon (tnixon@windows.microsoft.com), Technical Support. Private Correspondence.

[18] The Salutation Consortium, "Salutation Architecture Specification, Version 2.0c", June 1999, available at http://www.salutation.com.

[19] The Salutation Consortium, "Salutation Architecture: An Overview", Presentation, last accessed September 2004, available at http://www.salutation.org/tour/index.html.

[20] The Salutation Consortium, "Frequently Asked Questions," Last Accessed September 2004, http://www.salutation.org/faqs.htm#General%20Questions.

[21] N. Dulay, E. Lupu, M. Sloman and N. Damianou, "A Policy Deployment Language for the Ponder Language", Proceedings of the 7th IFIP/IEEE Symposium on Integrated Network Management (IM 2001), May 2001.

[22] Sun Microsystems, "Jini Architecture Specification, Revision 1.0," October 2000, available at http://www.sun.com/jini.

[23] Sun Microsystems: Jini Network Technology. "Jini Technology Architectural Overview." January 1999, available at http://wwws.sun.com/software/jini/whitepapers/architecture.html.

[24] UDDI . "The UDDI Technical White Paper", http://www.uddi.org

[25] UPnP Forum. "UPnP Device Architecture 1.0, v1.0.1." December 2003, available at http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf.

[26] Feng Zhu, Matt Mutka, Lionel Ni, "Classification of Service Discovery Protocols in Pervasive Computing Environments", MSU-CSE 02-24, Michigan State University, East Lansing, 2002.