Context Attributes: An Approach to Enable Context-awareness for Service Discovery

Choonhwa Lee and Sumi Helal
Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611-6120
{chl,helal}@cise.ufl.edu

Abstract

Service discovery problem has recently been drawing much attention from researchers and practitioners. Jini, SLP, and UPnP are among the few emerging service discovery protocols. Although they seem to provide a good solution to the problem, there is an unaddressed need of more sophisticated location and context-aware service selection support. In this paper, we introduce the concept of context attribute as an effective, flexible means to exploit relevant context information during the service discovery process. Context attributes can express various context information including service-specific selection logic, client, and network condition. We describe our approach and implementation, and present the experimental results of our context-aware service discovery implementation.

Keywords: *Dynamic service discovery, context attribute, context-aware service selection, mobile clients.*

1. Introduction

Mobile and wireless computing is permeating the globe, affecting the way we live and conduct business using portable computers such as laptops, PDAs, smart-phones, and even wearable computers. In becoming mobile, users innocently expect and demand the same computing luxury they used to (and still) enjoy in the fixed computing environment. Unfortunately, network resources (e.g. printers, fax machines, and file systems) and applications, collectively called *services*, do not follow the mobile users when they leave their offices or homes, or when they relocate to another location. Impromptu access to the services in the new environment is enabled by service discovery protocols [1] [2] [3] [4] [5] [6], which are re-shaping the way software and network resources are configured, deployed, and advertised.

Each service discovery protocol has adopted different service description models: Java service interface and assistant attribute objects of Jini, string-based attribute-value pairs of SLP, and XML descriptions in case of UPnP. Although there are pros and cons of their design decisions, e.g., their query efficiency and expressiveness, their service advertisements and queries are able to capture only static aspects of context: for example, a *server load* attribute can indicate the load on a server machine at the moment of service announcement but not the one at the time of service query later, because it is dynamically changing.

Service matching based on those declarative, static service descriptions provides minimal service discovery and filtering, leaving the rest of the work to users' manual selection; they have to try discovered service instances one-by-one, until they find an instance with satisfactory quality of service. But it would be a painful process if their mobile devices do not have enough computing and network resources needed for the manual selection. The problem gets even worse, when service population is large; image the future ubiquitous computing world where proximity, local, and global service discovery systems are bridged together. It is possible for them to be stormed by an unmanageably large number of service instances returned as a result of their queries, most of which turn out to be useless.

Existing service discovery protocols miss out on the opportunity of enormous benefits by exploiting useful context information relevant to service discovery. According to the context-aware computing literature [7], context is defined to be "any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object". The context information enables the right services to be delivered to the right users at the right time and place via refined service selection and recommendation, which is not attainable by declarative

service description models. In this paper, we present our work to incorporate the context-awareness into a general service discovery framework. It will allow us to benefit from implicitly captured context information once not sufficiently utilized.

The rest of the paper is organized as follows. Section 2 describes our context attribute approach for context-aware service discovery. Also, a service discovery scenario is given to illustrate how and what context information could be used for dynamic service discovery. Our implementation details and experimental results are presented in section 3 and section 4, respectively. Finally, section 5 includes related work and further discussions.

2. Context-aware service discovery

We first take a look at Jini, because our work is based on Jini service discovery framework. Then, we discuss what are missing in Jini, when it comes to dynamic, mobile service discovery.

2.1. Jini and mobile service discovery

Jini [1] [2] is built on top of Java object and RMI system. As illustrated in figure 1, a service proxy object is registered with a service registry, called a lookup service, which constitutes announcement process. Then, a client downloads the service proxy through discovery process and invokes some method on it to access the service.

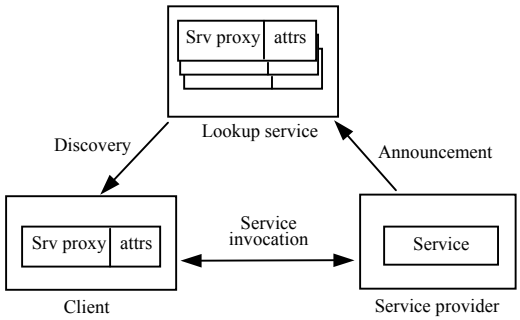

Figure 1 - Jini service discovery protocol

The lookup service can be thought of as a directory service, in that services are discovered and resolved through it. The services are to the client a Java interface, including methods that she will invoke to use them, along with associated descriptive attributes. The lookup service maps the interface exposed to the client to the set of service proxy objects. More specifically, performing a lookup by a service interface results in service proxy object(s) being downloaded to the client, which are actually RMI stubs that may communicate back with the services. But Jini's requirement for JVM and RMI appears to be an obstacle to its wide acceptance by small, mobile devices, although this problem has been alleviated by the introduction of Jini Surrogate architecture [8].

Context-aware service discovery means to provide the most appropriate service(s) to mobile users by exploiting any meaningful contextual information. Jini is chosen as base system for our work to build and demonstrate context-aware service discovery system. Like other service discovery protocols, Jini does not do much for context-awareness support. As to where to put the intelligence to capture context information, we argue that context-awareness should be supported by the infrastructure (i.e., service registries), since it is often the case where resource-constrained mobile devices can't afford it. Therefore, a basic assumption of our approach is that a client interacts with the closest service registry so that the context captured by the registry can approximate that of the client.

The coverage of Jini service discovery protocol is limited up to IP multicast range. This problem can be dealt with by building up a hierarchy of service registries to cover wider area, while keeping location- awareness [9] [23]. The Lincoln tunnel service of Jini Rio project [10] also addresses the need for the federation of Jini lookup services. Also, the integration of local and wide-area service discovery system was proposed by several research efforts [11] [12] [20]. Apparently, going beyond local areas, the importance of the right service discovery gets magnified. It is because users will likely to experience much broader range of service quality in wider areas, depending on which service instances are selected. As a flexible, effective means to capture context information including service quality, we introduce the concept of context attribute.

2.2. Context attribute

Existing service discovery frameworks do not sufficiently exploit context information for dynamic service discovery. For example, there's no support to figure out how far service providers and clients are away. In addition, there exists other information such as service quality that is inappropriate or impossible to be handled by static, declarative service descriptions. How can it tell if a particular service instance is better than any others in a particular location at the moment? Our approach to this problem is the context attribute shown in figure 2.

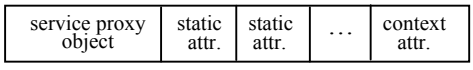

Figure 2 - Service record and context attribute

The *context attribute* is a special kind of attributes that are parts of service announcements. It is also called *dynamic attribute* in that its actual value is dynamically determined at the time of lookups, compared to *static attribute* that has a fixed value set at the time of announcements.

Services are first matched against a user query based on the static attributes, which produces a candidate service set. Then, through the context attribute evaluation, a lookup service further reduces it to a smaller set of qualified services to be returned to clients. In other words, services are ranked according to the evaluation results to ensure that the clients are given the best service instance(s).

It should be noted that clients are unaware of the existence of context attributes so they are not involved in the context evaluation at all. Service authors provide appropriate context attributes for their services and our context-aware service registry simply evaluates them without any knowledge of their internals (Context attribute is required to implement a predefined Java interface as explained in the next section.) The rationale behind this is that the service authors know what selection criteria matter to their services the most. Also, lookup services can be relieved of the burden to know the selection criteria of all types of services.

Benefits from exploiting context information for service discovery through the context attributes, which is a key feature of our context-aware, enhanced service discovery architecture, are summarized as follows:

- It provides a general, flexible means to enable sophisticated service selection. Service authors are allowed full flexibility to express selection criteria specific to their own services.
- The context attribute is an effective indicator of service quality that is able to capture various kinds of service qualities, including communication-related and application-related aspects.
- The evaluation cost of context attributes is amortized over multiple clients. To justify the non-negligible overhead of late evaluation, evaluation result is cached so that it can be reused for future requests for the same service, made by other clients.
- It is transparent to clients. The context attribute and its evaluation happening behind the scene are completely hidden from clients.

2.3. A service discovery scenario

In this sub-section, we present one service discovery scenario on the first day of a new student on UF campus to illustrate the merit of context attributes over static attributes. Table 1 summarizes 5 services used in the scenario.

Service	Static attribute	Context attribute
Guide	Manual selection	Closeness detection based on domain name
Movie preview	Manual selection	Network bandwidth and delay detection
Movie theater	Random selection	Physical distance understanding
Printer	Constant load information pushing	Load information pulling when necessary
Bus schedule	Flag attribute	Advertisement controlled by time of the day

Table 1 – Static attributes vs. context attributes

2.3.1. Guide service. A new student arrives at UF campus to start her study. At the entrance of a residence hall, she picks up her PDA to discover area guide services. Assume that there are 3 guide services registered with a service registry around: *dorm guide*, *campus guide*, and *Gainesville city guide*.

Context attribute: A *domain* context attribute attached to the guide services determines the closeness between the service registry and service instances based on DNS domain name. Since the service registry is in *dorm.ufl.edu* domain, the *dorm guide* is given the highest score as a result of the evaluation. The student gets back a service instance list sorted in the order of *dorm guide*, *campus guide*, and *city guide* service.

Static attribute: The user has to browse the list to figure out which service instance she needs, relying on descriptive static attributes. Unfortunately, it may take too long time until she finds that it doesn't provide detailed information about the residence hall, if she first chooses the *campus guide* service.

2.3.2. Movie preview service. She finished her check-in process. Now, she wants a movie for the rest of the day. She needs to check movie previews from movie-ad sites, using a MPEG player program. They are likely out-of-state sites (i.e., remote services) maintained by nation-wide movie distributors.

Context attribute: The best site in terms of network bandwidth and delay can be determined by *ping* context attributes which check network condition for advertised sites.

Static attribute: She will pick up one at random. If she selects a service with intolerable QOS, she will suffer from long delay and jitter.

2.3.3. Movie theater service. At last, she made up her mind on what movie to watch. Now, she needs to search for local theaters that advertise themselves to the service registry.

Context attribute: The nearest one is recommended by a *location* context attribute based on ZIP codes. Assume

that the registry is preconfigured with its own ZIP code or the context attribute can somehow acquire it. Then, the service registry, more specifically the location context attribute, will be able to figure out approximate distances to the theaters from its own ZIP code and theaters' ZIP codes.

Static attribute: She has to make a random selection of ZIP codes (i.e., part of service descriptions displayed on her PDA). But she doesn't know what her ZIP code is, since it is her first day on the campus.

2.3.4. Printer service. She wants to print out the direction to the theater using a near-by printer. Note that some printers are being heavily used by housing staffs to process student check-ins. According to the movie schedule from the theater service, she knows that she doesn't have much time. She needs to rush.

Context attribute: A *load* context attribute is used to figure out which printer is least loaded. The load attribute connects back to its service for current queue length, which returns up-to-date load information in response. Then, the service registry is able to present to the user the list of printer services sorted in the ascending order of queue lengths.

Static attribute: A declarative, static load attribute is associated with a printer service. The attribute should be updated immediately as soon as the printer load changes, although nobody requests the service. This results in the waste of valuable network resource and processing power. Also, the user has to make a manual selection based on the advertised load information.

2.3.5. Bus schedule service. She decides to take a bus. The access point device at a bus stop in front of her residence hall hosts a service registry. Each bus running through the bus stop advertises itself to the registry.

Context attribute: The bus services registered there are made either shown or hidden depending on the evaluation results of their *in-service* context attributes. In other words, buses will not be returned to the user according to the context attribute evaluation results, if they already stopped running for the day.

Static attribute: Each bus service may have an *in-service* flag attribute to indicate whether or not the bus is still in service for the day and the attribute is to be changed every night and every morning.

3. Implementation details

Our prototype is built by adding context-awareness processing to Sun's Jini reference implementation, named *reggie*, version 1.2. All changes we made are kept within the Jini lookup service and standard Jini API's are not affected. Therefore, our modifications are transparent to clients; our augmented service registry can support services either only with ordinary static services or with static and context attributes. Services with context attributes are given special treatments by our enhanced service registry but the clients are kept unaware of it.

3.1. Definition of context attribute

Extending a Jini naming space *net.jini.lookup.entry*, we added several Java classes related to the context attribute, including *LDContextEvaluator, RDContextEvaluator, AbstractLDContextEntry, AbstractRDContextEntry, ContextEvaluation.* Parts of these class definitions are shown in figure 3, 4, and 5.

```
public interface LDContextEvaluator {
    ContextEvaluation evaluate();
}

public interface RDContextEvaluator extends Remote {
    ContextEvaluation evaluate() throws RemoteException;
}
```

Figure 3 - Java interface definitions of local and remote context attribute

```
package net.jini.lookup.entry;

import java.io.Serializable;

public class ContextEvaluation implements Serializable {
    long validUntil;       // duration + current time.
    long duration;         // period during which this
                           // evaluation remains valid.
    long contextIndex;  // indicates service quality.

    ....
}
```

Figure 4 - Java class definition of context evaluation result

A context attribute is either a local or remote object. The local context attribute and remote context attribute implement *LDContextEvaluator* and *RDContextEvaluator* interface, respectively. If the evaluation can be performed by a Jini lookup service alone, it is a local context attribute. For example, if a context attribute measures network hop counts to its service instance, it is a local object evaluated solely by the lookup service. It does not involve the corresponding service instance in the evaluation process. As an example of a remote object, we can think of a context attribute that probes current load on the server machine. In this case, the context attribute will be a stub object that makes a RMI call back to its service instance.

All context attributes are required to implement either of the two interfaces, so our lookup service can evaluate them by simply invoking their *evaluate()* methods without understanding their internals. The *evaluate()* method returns a *CxtEvaluation* instance that contains *duration*, *validUntil* and *contextIndex*. The *duration* field indicates a time period during which evaluation result is likely to remain valid. This allows the evaluation result to be cached in our lookup service for future requests from local clients at the site. The *validUntil* field is an absolute time when the cached result expires. The third field, *contextIndex,* represents a comparative quality index of services in question in a numeric form, for example, a normalized index out of 100. It is important to know that this normalization is defined on service type-by-type basis, as each Jini service interface is autonomously defined by a community of all parties involved in the service type. This ensures that relative superiority can be indicated by a normalized number among service instances of the same type. In other words, service selection criteria specific to a service type are defined as part of the service standardization and the index value will be valid across the instances of the service type. Therefore, the concept of *contextIndex* frees our lookup services from having to understand the normalization semantics of every service type for comparison. We argue that it is well aligned to Jini design philosophy promoting the autonomy for individual service implementations.

```
public abstract class AbstractLDContextEntry extends
        AbstractEntry implements LDContextEvaluator {
    public ContextEvaluation val;
    ...
}

public abstract class AbstractRDContextEntry extends
        AbstractEntry implements RDContextEvaluator {
    public Object rObj;
    public ContextEvaluation val;
    ...
}
```

Figure 5 – Base class definitions for local and remote context attribute

Figure 5 shows base classes for local and remote context attribute. They extend *net.jini.entry.AbstractEntry* class that is a base *Entry* type for Jini service attributes. By subclassing it, service authors are relieved of the internal details of context attributes. Both contain a *ContextEvaluation* type *val* which caches their previous context evaluation results. Although not shown in the figure, the constructor of *AbstractRDContextEntry* class exports itself to RMI runtime system so that a lookup service can make a connection to it for context evaluation. Finally, figure 6 shows a sample remote context attribute, *ServerLoad* attribute.

```
import java.rmi.RemoteException;
import net.jini.lookup.entry.AbsractRDContextEntry;
import net.jini.lookup.entry.ContextEvaluation;

public class ServerLoad extends AbstractRDContextEntry {
    public ServerLoad() throws RemoteException {}

    private long getSystemLoad() {
        // get current load and return it
    }

    public ContextEvaluation evaluate()
                        throws RemoteException {
        long sysLoad = getSystemLoad();
        return new ContextEvaluation(1*60*1000,
                                MAX_LOAD – sysLoad);
    }
}
```

Figure 6 – A sample ServerLoad context attribute

3.2. Context attribute processing

The context-awareness added to original Jini lookup services is transparent to clients, since, provided by service authors, context attributes are evaluated by our lookup service (and service providers in case of remote attributes). Therefore, the clients make a discovery request specified in Jini service interface and static attributes without worrying about whether they are seeing original lookup services or our enhanced lookup services. In other words, original Jini API's are kept untouched.

```
ServiceMatches lookup(ServcieTemplate tmpl,
                                        int maxMatches) {
    int candidateSetSize = maxMatches * CANDSETSIZE;
    candidates = slookup(tmpl, candidateSetSize);

    for (each instance in candidates.items) {
        if (it has an instance of AbstractLDContextEntry
                            or AbstractRDContextEntry) {
            if (previous evaluation expired)
                evaluate the context attribute;
        }
    }

    sort the candidate set according to contextIndex
    remove context attributes from each service;
    return top maxMatches instances;
}
```

Figure 7 - Context attribute processing

As shown in figure 7, our lookup service considers *CANSETSIZE* times the number of service instances a client requests. The *slookup(),* the original *reggie*'s lookup method, is first performed based on the static query. Then, it goes through each service instance to see if any context attribute attached to it needs to be re-evaluated. Again, previous evaluation result is cached in the *val* field of the attribute. A thread is assigned to each context attribute so that the evaluations can be performed in parallel. But each attribute may take different evaluation time, so we need to set a timer. When this timer expires, any ongoing evaluation is aborted. After then, it sorts the candidate set in the descending order of *contextIndex* value. The service instances with context attributes are favored over those without them. Note that some instances may not have any context attribute attached to them, even if they are all the same service type. Before returning top *maxMatches* instances to its client, context attributes are removed from their *Entry* list to make the context awareness processing hidden from the client.

We have also developed a hierarchy of Jini lookup services to support non-local service discovery. Inter-registry advertisement/discovery and lease management have been implemented. Services can be propagated to parent lookup services up along the hierarchy. Similarly to *LDContextEvaluator* and *RDContextEvaluator*, we introduced *LAContextEvaluator* and *RAContextEvaluator* for service advertisements. Two abstract base classes, *AbstractLAContextEntry* and *AbstractRAContextEntry*, were added as well. These context attributes for advertisements are evaluated at the time of registration. On registration requests from a service, our service registry passes it on to its parent registry if it has an attached advertisement context attribute. Depending on the evaluation result by the parent, the service may be further propagated or dropped at that point. In other words, the context attributes prescribe pre-conditions that should be met to further propagate. This way service providers can control the reach of their service advertisements so that their resources can be dedicated to their targeted clients by avoiding being disturbed by unintended customers. Otherwise, clients won't find that the services are not for them, until they are rejected by the service access control logic somehow.

4. Experiments

We conducted a series of experiments of our prototype context-aware lookup service based on Jini reference implementation version 1.2 from Sun. The experiments consist of two parts: service registry side and client side performance experiments. We used a Sun Ultra-60 workstation with 512M RAM, running J2SE v1.3.1_02.

The benefits from the context-awareness are evident. But, at the same time, it introduces processing overhead to the lookup service that must be scalable to support a large number of service registrations. Jini service attributes (i.e., *Entry* objects) are packaged as serialized bit streams (i.e., *MarshalledObject*) so service matching can be made via bitwise comparison. It allows fast processing by lookup services without reconstituting the bit streams into real objects. This is why Jini can support only exact matching for service lookup. But in our case, the *MarshalledObject* objects for context attributes should be reconstituted back into objects to invoke *evaluate()* methods on them. Additional processing burden and network traffic will be introduced by deserializing the marshaled objects and the following *evaluate()* method calls. The first set of experiments is designed to address this scalability concern.

Our lookup service running on a Sun Ultra-60 machine is populated with 10,000 service instances of the same type. The services used for experiments have two attributes; one is an instance of *net.jini.lookup.entry.ServiceInfo* class and the other is an instance of one of *net.jini.lookup.entry.Location*, *Domain*, *Load*, or *Ping* class. Note that the *ServiceInfo* and *Location* instance are static attributes and other three are context attributes. The *Domain* context attribute looks into the DNS domain name of the registry, which may be used by service providers who want to publish their services within a certain domain boundary. The *Load* attribute connects to its service provider to ask current load on the server machine. Finally, round-trip time is measured by the *Ping* attribute that may be found useful by network latency sensitive services. The *Domain* and *Ping* attributes are local context attributes, while the *Load* attribute is a remote context attribute.

For each of the above services, a client on the same machine makes three discovery requests to the lookup service with 10,000 instances of that type, asking for 1, 10, and 100 instances each time, although the queries are satisfied with all 10,000 instances. For each query, we measure CPU time taken by our augmented *reggie* using a Java profiling tool [13]. The evaluation result cache feature is disabled for these experiments.

Figure 8 shows CPU time for each service type. The lookup service performs well for the *Location* static attribute, while processing time for the three context attributes rises, as the number of service instances increases, especially for 100 instances. But it should be noted that this is an extreme setting for scalability experiments and it is unlikely to happen in a real situation, unless a client browses the whole services in

the lookup service (in this case, context attribute processing would be bypassed.), for the following reasons; candidate services are first selected by static attribute matching and then, context attributes are evaluated for those services in the candidate set. In addition, the cache of previous evaluation will reduce the number of service instances that need to be evaluated. Also, as expected, the local context attributes (*Domain* and *Ping*) take less time than the remote attribute (*Load*).

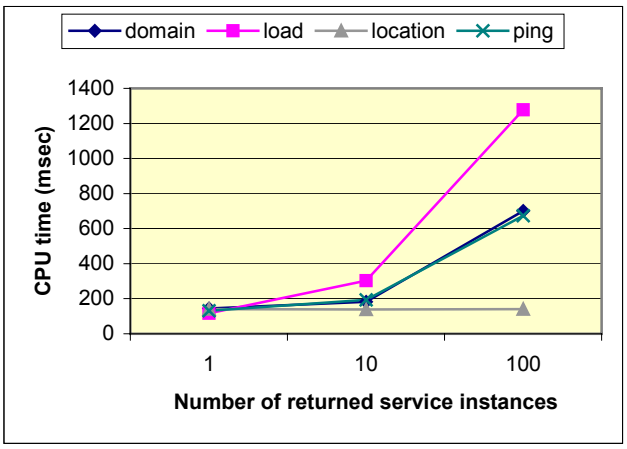

Figure 8 - Jini lookup service performance

Our second experiment set is to analyze contribution to the CPU time for the *Ping* context attribute. Again, the cache is disabled for these experiments. Sun's *reggie*, uses *com.sun.jini.thread.TaskManager* to manage a pool of threads for Jini discovery and event handling. We use another *TaskManager* instance for threads to evaluate context attributes. Figure 9 shows contribution to CPU time for the *Ping* attribute evaluations.

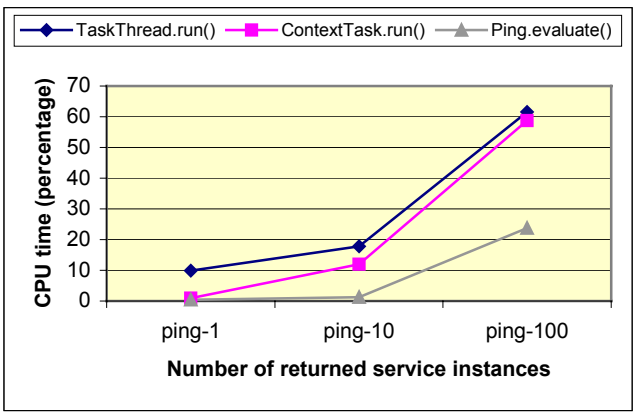

Figure 9 - Composition of lookup service CPU time

The plot shows relative time spent on three methods. The *TaskManager's TaskThread* calls our *ContextTask.run()* that in turn, executes *Ping.evaluate()* method. *TaskThread* also manages other tasks such as Jini discovery and event task. Since *ContextTask* time is part of *TaskThread* time and *Ping.evaluate()* time is part of the *ContextTask* time, each line in the graph subsumes its underlying line. As the number of returned services increases, a larger portion of CPU time is taken by *TaskThread*. At the "ping-100", 23 % of the CPU time is spent on *Ping.evaluate()* method, while 35 % of the time was wasted for thread synchronization.

To measure time observed by clients, we run another set of experiments; for three services including *compute*, *dictionary*, and *ftp* service, we measure lookup time and service time. The *compute* service offers a computation service that employs the *Load* context attribute in the previous experiments. The *dictionary* service returns the meaning of a word asked by clients, similarly to dictionary web services. Since latency is the most important for this type of interactive services, it uses the *Ping* attribute used in the previous experiments. Finally, the *ftp* service uses a new *Bandwidth* attribute that examines available bandwidth between the lookup service and the ftp service provider by rehearsing a 100K packet. For each service type, we populate 8 service instances with different service quality ranging from *1x, 2x, 4x, 8x, 16x, 32x, 64x,* to *128x*. Figure 10 shows time for lookup() method by a client under these experiment settings.

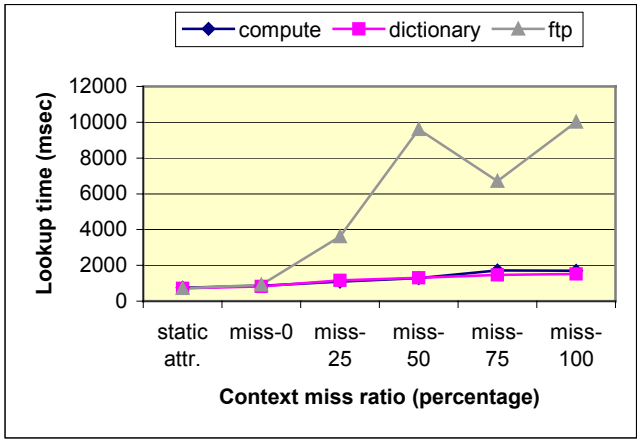

Figure 10 - Client lookup time

The x-axis represents different context evaluation miss ratios. For example, "miss-0" means that cached context evaluations are valid for all candidate services produced by static attribute matching. In addition, the case of "static attribute only" is shown for comparison. As the miss ratio increases, the client experiences

moderate increases of lookup time for *compute* and *dictionary* services. But for *ftp* service, the lookup time changes drastically. This is because average context evaluation time and variation vary across service types. Exercising a 100K packet over the *1x* bandwidth link takes much longer time. This is why we need a timer for context evaluation mentioned in sub-section 3.2. Also, we can see that links with better bandwidth for "miss-75" case than for "miss-50" case have happened to be chosen.

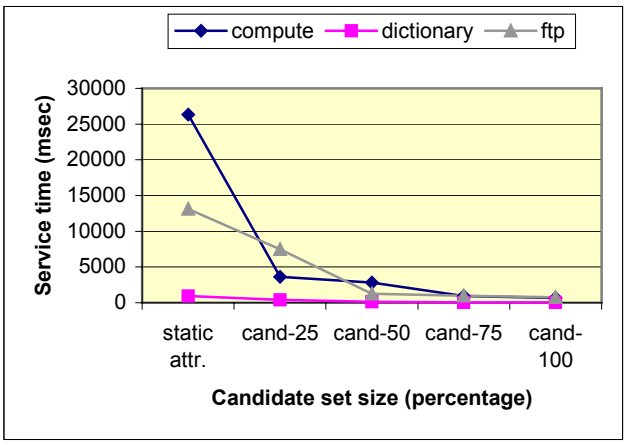

Figure 11 - Service time experienced by client

Figure 11 shows time for the first round of service uses. For example, service time for *ftp* service is the time to fetch 1M data. The x-axis represents how many cached evaluations are valid at the time of client requests. For example, "cand-50" means that context evaluation results are available for 4 out of 8 services. Service selection is made among those cached services without evaluating expired ones. For static attribute, the service time comes close to the average of all *1x* through *128x* 8 instances, since the instances are selected at random fashion. The graph shows that service time can be dramatically reduced by service recommendation via context attributes at "cand-25". The service time continues to improve, as the candidate set size grows towards the perfect case of "cand-100".

5. Related work and discussion

Our work on designing and developing a prototype for context-aware service discovery framework has been influenced by several emerging fields of research and technologies, including service discovery, server selection, and context-aware computing technology. The server selection problem is concerned about a particular type of services, e.g., selecting the best Web server among identical replicas. In contrast, the service discovery problem is to locate general, all possible types of services.

Various server selection works show a full spectrum ranging from server side to network and to client side approaches [21] [14] [22] [15] [16]. However, most of them lack generality to be used by a variety of applications, since they are fine-tuned to a specific type of services or a specific application domain. To overcome this limitation, the server selection schemes are being generalized by recent RSerPool IETF drafts [17] [18]. It is more concerned with communication-oriented aspects than service-oriented aspects, which positions itself as an additional entity separate from general service discovery framework. It contrasts our approach, i.e., a general service discovery framework that is also capable of supporting various server selection mechanisms through context attributes. By integrating the two into a single framework, our architecture provides a single-step service discovery and selection.

Most of service discovery protocols adopt declarative, static service description models. While it allows efficiency and simplicity for service query resolutions, it is not powerful enough to capture context relevant to service discovery. Infrastructure support for value-added service discovery is also found at [19] to propose *Selection* and *Sort* extensions to SLP. Their approach is to rank service instances according to static-attribute-based sort key list specified by a client. It means the client must understand what attributes for the service type matter the most to be able to specify the selection criteria. Besides, the selection process is solely performed by service registries without any help of servers, which may limit context information that can be captured. The idea of the context attribute draws from client-side server selection mechanisms, e.g., Smart client [15], but it is generalized for service discovery in our work, while they assume the Internet services hosted on a cluster of workstations. Unlike others, our approach is able to capture dynamic aspects of context to enable much more sophisticated selection. Consequently, it achieves both the specialty of server selection mechanisms and the generality of service discovery protocols.

We implemented the context-awareness support on Jini lookup services with resource-constrained mobile devices in mind. However, it can be brought down to client devices, if they have enough resources to handle the context attributes. Then, more accurate context information would be able to be captured for service discovery and selection. Also, there would be no concern about lookup service scalability, since the context attributes will not be interpreted by lookup services. But a downside is that context evaluation results can't be shared among clients.

6. Conclusion

The *context attribute* just defines a general agreement and its real implementation is left to service authors. Therefore, it is an effective means to capture relevant, dynamic context related to client, service, and network condition in between them, in connection with dynamic service discovery. Since it provides a framework for any service-specific selection logic, the context attribute achieves both the specialty of server selection mechanisms and the generality of service discovery protocols.

We have developed a prototype of context-aware service discovery framework by building context-awareness support on Jini. Also, we have evaluated our implementation through a series of experiments to measure the overhead and gain on lookup service and client side.

7. References

[1] "Jini Technology Architectural Overview," January 1999, http://www.sun.com/jini/whitepapers/architecture.html.

[2] "Jini Specifications v1.1," October 2000, www.sun.com/jini/specs.

[3] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, Version 2," IETF RFC 2608, June 1999. http://www.ietf.org/rfc/rfc2608.txt

[4] Erik Guttman, Charles E. Perkins, and James Kempf, "Service Templates and Service: Schemes," IETF RFC 2609, June 1999. http://www.ietf.org/rfc/rfc2609.txt.

[5] Universal Plug and Play Device Architecture, http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[6] Salutation Consortium, "Salutation Architecture Specification Version 2.0c," June 1, 1999, http://www.salutation.org/specordr.htm.

[7] Anind K. Dey, Gregory D. Abowd, and Daniel Salber, "A Context-Based Infrastructure for Smart Environment," In Proceeding of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)

[8] Jini Surrogate project page, http://surrogate.jini.org.

[9] Rui Jose and Nigel Davies, "Scalable and Flexible Location-Based Service for Ubiquitous Information Access," In Proceedings of First International Symposium on Handheld and Ubiquitous Computing, HUC'99, September 1999.

[10] Jini Rio project page, http://rio.jini.org.

[11] Jonathan Rosenberg, Henning Schulzrinne, and Bernd Suter, "Wide Area Network Service Location," IETF Internet Draft, November 1997.

[12] Toyotaro Suzumura, Satoshi Matsuoka, and Hidemoto Nakada, "A Jini-based Computing Portal System," In Proceedings of SC2001, November 2001.

[13] ej-technologies' JProfiler, http://www.ej-technologies.com/jprofiler/overview.html

[14] Samrat Bhattacharjee, Mostafa H. Ammar, Ellen W. Zegura, Viren Shah, and Zongming Fei, "Application-layer anycasting," In Proceedings of the IEEE INFOCOM '97, April 1997.

[15] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler, "Using Smart Clients to Build Scalable Services," In Proceedings of the 1997 USENIX Annual Technical Conference, January 1997.

[16] Sandra G. Dykes, Clinton L. Jeffery, and Kay A. Robbins, "An Empirical Evaluation of Client-side Server Selection Algorithms," In Proceedings of IEEE INFOCOM '00, March 2000.

[17] Michael Tuexen, Qiaobing Xie, Randall Stewart, Melinda Shore, Lyndon Ong, John Loughney, and Maureen Stillman, "Requirements for Reliable Server Pooling," IETF Internet Draft, May 2001.

[18] Michael Tuexen, Qiaobing Xie, Randall Stewart, Melinda Shore, Lyndon Ong, John Loughney, and Maureen Stillman, "Architecture for Reliable Server Pooling," IETF Internet Draft, April 2001.

[19] Weibin Zhao, Henning Schulzrinne, Erik Guttman, Chatschik Bisdikian, and William Jerome, "Selection and Sort Extension for SLP," IETF Internet Draft, June 2002.

[20] Chatschik Bisdikian, John S. Davis II, William F. Jerome, and Daby M. Sow, "Emerging Research Opportunities in Ubiquitous Service Discovery," In Proceedings of New York Metro-area Networking Workshop (NYNET01), March 2001.

[21] Thomas P. Brisco, "DNS Support for Load Balancing," IETF, RFC 1794, April 1995.

[22] Craig Partridge, Trevor Mendez, and Walter Milliken, "Host Anycasting Service," IETF RFC 1546, November 1993.

[23] Rui Jose, Adriano Moreira, Filipe Meneses, and Geoff Coulson, "An Open Architecture for Developing Mobile Location-Based Applications over the Internet," In Proceeding of the 6th IEEE Symposium on Computers and Communications, July 2001.