ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Network and
Computer Applications 28 (2005) 233–248

Journal of NETWORK and COMPUTER APPLICATIONS

www.elsevier.com/locate/jnca

Towards a flexible service discovery

Tao Gu*, Hung Keng Pung, Jian Kang Yao

Network Systems and Services Laboratory, Department of Computer Science, National University of Singapore, 3 Science Drive, 2 Singapore, 117543

Received 24 December 2003; received in revised form 11 May 2004; accepted 4 June 2004

Abstract

To support m-commerce applications, a service discovery mechanism where services can announce their presence and mobile users can locate these services is needed. Service discovery in dynamic mobile environments poses many challenges such as service providers may create and delete services or servers anytime; mobile services may be deployed in various forms, etc. In this paper, we propose a design for a Service Locating Service (SLS), which addresses some of these issues to provide a flexible service discovery mechanism for m-commerce applications. In our architecture, we adopt a dynamic tree structure for organizing SLS servers to meet the dynamic requirements of services and servers; we introduce service aggregation for fast locating; and we also propose multiple service matching mechanisms, which contain an attribute matching engine and a semantic matching engine for different service interfaces. We describe our concepts, architecture and implementation, and present a performance study for our prototype.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Service discovery; Mobile services; Service aggregation; Semantic matching

1. Introduction

The emergence of wireless networks and mobile devices has created a new field known as mobile commerce in which applications and services are becoming accessible from user's network-enabled mobile devices. Recent researches (Sadeh, 2002; UMTS, 2000) show that mobile services are expected to expand significantly in the next few years in

* Corresponding author. Tel./fax: +65-68747381.
E-mail addresses: gutao@comp.nus.edu.sg (T. Gu), punghk@comp.nus.edu.sg (H.K. Pung), yaojiank@comp.nus.edu.sg (J.K. Yao).

1084-8045/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2004.06.001

term of popularity, variety and complexity. Examples of such services are applications that allow mobile users to locate and download MP3 music from a music server. With numerous mobile services and devices, how to facilitate users in discovering these services is indeed a challenging task judging from the diversity of services and the dynamics of users as well as service providers.

The design of a service discovery mechanism for m-commerce applications poses a number of challenges. First, mobile services are dynamic as service providers may create, update, and change them at any time. A server in which services are stored may also join or leave the system frequently. To meet the dynamic requirements of services and servers, the Service Locating Service (SLS) servers in our system are constructed and managed using a dynamic tree. The dynamic tree structure has the ability to quickly adapt to the changes imposed by creating, updating or deleting services and adding or removing SLS servers. It is able to re-configure itself quickly.

Second, how to locate a service in an efficient way is an important issue. For example, searching among different servers, which stores large number of services, may become very slow and inefficient. As an attempt to address this issue, we propose the concept of service aggregation, which allows the lower-tiered SLS servers to keep detailed service information and the upper-tiered SLS servers to aggregate this specific information into a more compact form. These aggregated information serve as routing index for a query to quickly find out a searching path that leads to the destination server. Hence, the efficiency of service locating can be improved.

Third, mobile services are deployed in various forms and with different service interfaces. A same service may be represented in an attributed form or a semantic description or Jini's interface, etc. A flexible service discovery system should enable to locate all available services conforming to a particular functionality or set of attributes. We incorporate both attribute-based and semantic matching mechanisms in our system and also retain Jini's interface matching to provide users a flexible means to search for services. The SLS system is able to convert a user's query between different forms to adopt different matching mechanisms for locating services.

The initial design of the SLS system was included in the paper (Gu et al., 2003). The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes our design concepts; followed by the SLS system architecture in Section 4. In Section 5 we present our implementation and performance measurements; and finally we conclude in Section 6.

2. Related work

In recent years, many service discovery architectures arising from both industrial research and academic research. Jini (Waldo and Arnold, 2000) is a distributed service discovery architecture built on top of Java object and RMI (Java remote method invocation) system. A service proxy object is registered with Jini Lookup Service. A client downloads the service proxy and invokes it to access the service, which is identified by means of Java class hierarchy. Jini employs Java interface matching. As such, client is solely responsible for knowing the precise name of the Java class representing the service.

The Jini architecture has the limitation on scalability as it does not provide any solutions to connect Jini federations, which may reside in global networks. Universal Plug and Play (UPnP, 2000) has a Peer-to-Peer architecture based on TCP/IP networks and is designed to accommodate home networks or small office networks. It uses the Simple Service Discovery Protocol for discovery of services, which can operate with or without a lookup service in the network. It has a simple XML matching mechanism; however, XML was defined only at syntactic level. Salutation (The Salutation Consortium, 1999) is an open standard of communication independent service discovery and session management protocol. The Salutation architecture defines an entity called the Salutation Lookup Manager that functions as a service broker for services in the network. The services are discovered based on a comparison of the required service types with the service types stored in the Salutation Lookup Manager directory. Universal Description, Discovery and Integration (UDDI) (Bellwood, 2001) is an emerging industry standard that defines a business oriented discovery mechanism to a global registry holding XML-based WSDL service descriptions. It uses SOAP that allows one program to invoke service interfaces across the Internet in a language independent and distributed manner. UDDI aims at global networks, but it is targeted towards web services (Curbera et al., 2002) and has less dynamic support. Secure Service Discovery Service (SDS) (Czerwinski et al., 1999) is a research level service discovery system developed at University of California, Berkeley. It has a client-repository-server type architecture and has an XML-based semantic matching mechanism. Service descriptions and queries are specified using XML. It has simple semantic matching capability based on XML. International Naming System (INS) (Adjie-Winoto et al., 1999) is a resource discovery and service locating system for dynamic and mobile networks developed at Massachusetts Institute of Technology. It uses a simple naming language based on attributes and values to achieve expressiveness, integrates name resolution and message forwarding that tracks change, and uses soft-state name discovery protocols that enable robust operation. INS has the limitation when it scales to large numbers of resources spread throughout wide-area networks. INS/Twine (Balazinska et al., 2002) achieves more scalability by partitioning the name space across resolvers by mapping names into numeric keys.

3. Design concepts

In this section, we discuss some of the major concepts used in our architecture.

3.1. Dynamic tree structure

In our system, we adopt a dynamic tree structure for organizing SLS servers. The tree is dynamically constructed in multiple levels. SLS servers can be added to or deleted from any levels of the tree during runtime. Whenever a SLS server joins or leaves the tree, the SLS system responds to these changes by updating and notifying its parent or children servers. We will describe more details in Section 4.2. Arranging the SLS servers in multiple levels also enable us to apply a scaleable service aggregation scheme.

3.2. Service aggregation

The general concept of data aggregation refers to any process in which information is gathered and expressed in a summary form. This technique has been used in many areas such as statistical analysis, etc. Based on this technique, we introduce the concept of service aggregation, which allows each SLS server to keep summarized service information about all its children servers. In this way, each server stores these aggregated service information as routing index to all its children servers. The main benefit of this scheme is that the searching paths can be found quickly based on the routing table in each server. Service aggregation can also minimize the storage cost at each SLS server as the aggregated information of all its children servers is represented using a compact form. Furthermore, it can reduce the cost of message exchange since only aggregated service information is transmitted between a pair of parent–child servers. The concept of service aggregation was implemented by an aggregator (as described in Section 4.4.2).

3.3. Multiple service matching mechanisms

Mobile services are commonly developed by different service providers using different service creation platforms; and hence represented in various forms. Most existing mobile services are defined in syntactic level such as predefined service types and attributes in Salutation or Jini's interface; and they leverage on attributed-based or interface-based matching mechanisms. We believe that syntactic level matching is not adequate as the same service may be implemented by different interfaces. There is a need to discover services in a semantic manner, which based on the capabilities and functionalities of services. Semantic matching plays a key role in discovering mobile services due to the heterogeneity of service interfaces in mobile environments.

With the emerging trend of the Semantic Web, services can be defined in a semantic manner by using semantic languages. For example, the DAML initiative is developing a DAML-based Web Service Ontology named DAML-S (Ankolenkar et al., 2002) to describe Web Services based on their capabilities and functionalities. Clearly the technology for the semantic web is also applicable to semantic service discover in a mobile domain. We have designed an ontology for describing mobile services by tailoring DAML-S for the special need of mobile services such as lightweight—the ontology has to be light weight to minimize information load and communication cost; context constraints—mobile services are often constrained with the capability of mobile devices and network condition, etc.

In SLS, we introduce an attribute matching mechanism when services are described in syntactic level by attributes and a semantic matching mechanism when services are described in semantic level using semantic language-DAML + OIL (Horrocks, 2002). We also retain Jini's interface matching employed by Jini infrastructure. Service providers may register a mobile service using different service interfaces such as a service template describing attributes, a DAML + OIL (DAML in short) description of the service, or Jini's interface. Clients attempting to discover services will have the following service matching options: either to invoke a specific matching mechanism or invoke other service matching mechanisms. The client's query can be converted between different interfaces. For example, when a client issues a query described in an attributed-based template, the corresponding attribute-based matching mechanism will be automatically invoked. The client can also request the system to convert his query to other forms—semantic representation or Jini's interface to invoke semantic matching and Jini's interface matching. Hence, the multiple service matching mechanisms provide users a flexible means to discover mobile services.

4. System architecture

The SLS system consists of three components: SLS servers, SLS clients and services as shown in Fig. 1. A SLS server is a service information repository, providing SLS clients with access to all available services. A Service can are registered to any SLS server with different service interfaces, i.e. using a service template, a DAML description or Jini's interface. SLS clients on behalf of end users can search for services. In the following section, we describe these components in details, focusing on the roles in the system and how they interact with each other.

4.1. SLS communication

For communication between a pair of parent–child SLS servers, between SLS clients and servers, and between services and SLS servers, we use Java RMI. The main advantage

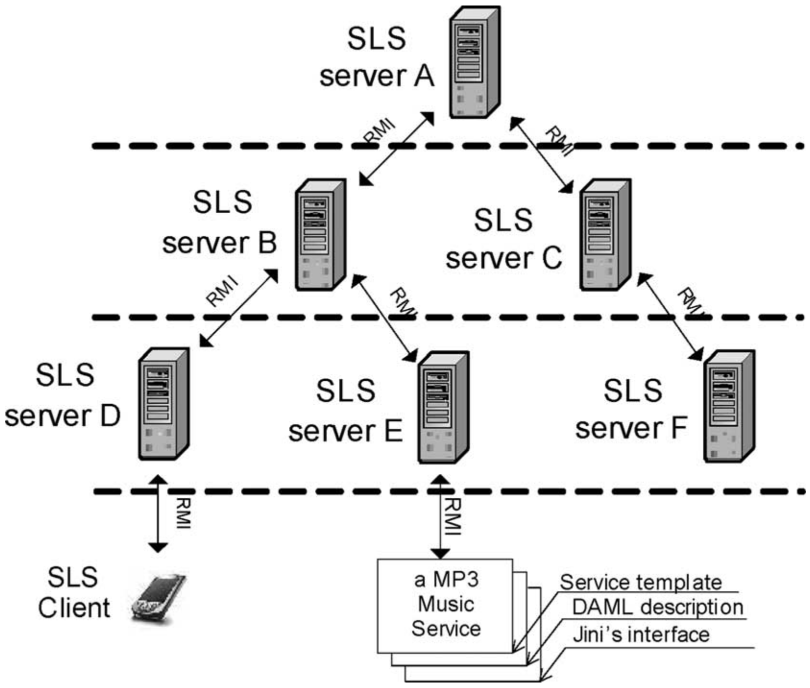

Fig. 1. An overview of the SLS system architecture.

of Java RMI is interoperability between heterogeneous platforms; and it also offers a certain degree of security features. RMI allows distributed objects can invoke each other's methods even if such underlying platforms as processors, operating systems, and programming languages are heterogeneous. This feature fits the need of m-commerce applications well due to heterogeneous platforms in such a domain. The requirements for implementing an RMI client or server are simply having an implementation of JVM; and it is easily met giving evidence that many mobile devices with such capability are available nowadays.

4.2. Building a dynamic server tree

To construct a server tree, we take a bottom-up approach. Starting up a new SLS server requires performing self-configuration and sending advertisement messages on global multicast channel. These messages contain the multicast address to use for sending service announcements. As shown in Fig. 2, when the service load on Server A reaches a certain threshold, one or more new servers (B or D) on the same level are created and configured. The system requires setting up a new parent Server C at one level up to connect these children servers. The abstracted service information in Server A, B and D will be

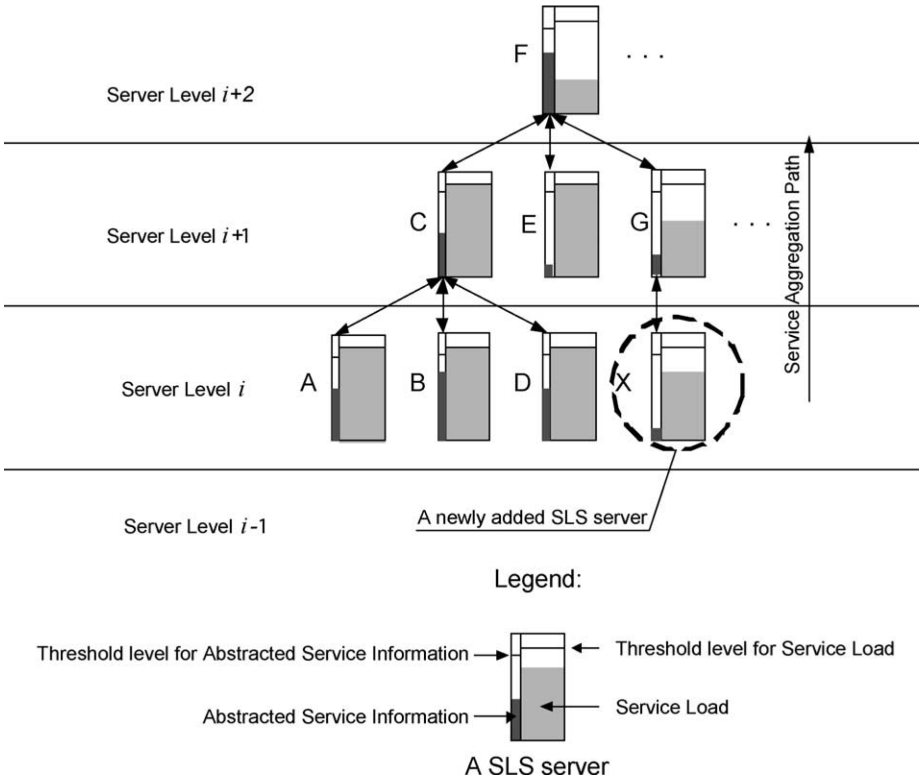

Fig. 2. The dynamic SLS server tree.

aggregated to C. The same process is repeated if the service load on Server C reaches a threshold. The path for service aggregation is always from low-level servers to high-level servers. A parent server keeps track of their children servers through the SLS server manager (described in Section 4.4.1).

To allow the flexibility of server joining and leaving, a new server can select any server as its parent to join as long as the abstracted service information on its parent server is below the threshold. For example, when Server X has found a Server G using multicast or unicast and wishes to join the tree, the abstracted service information in Server X will be transmitted to Server G. These abstracted service information will be propagated upwards along the child-to-parent path until the root server is reached. Our system also allows joining two server trees, which may belong to different service providers. For example, a server tree may join the existing tree (A–G) by connecting its root server to Server G (i.e. Server X is replaced with a server tree). Again, the same service aggregation process is applied. Any SLS server or a set of SLS servers can also leave the tree. For example, if Server G wants to leave the tree, a service aggregation update will be preformed along the child-to-parent path and all its children servers (i.e. X) will have to find another parent server to join. In the case of server failure, in our current design, we treat the failed server as a deleted server. Hence, all its children servers need to find a new parent server to join.

The basic function of the SLS is to answer clients' queries. A SLS server uses its Service Locating Manager, which consists of multiple service matching engines (described in Section 4.4.3) to search for services. A service can be registered to any server based on server load, geographic region, administrative domain, and favourite, etc.

4.3. Services and SLS clients

Services in m-commerce applications can be described and registered to a SLS server in various forms in our system: a service template, a DAML description or Jini's interface.

A service template consists of attribute-value pairs describing the properties of a service in terms of service type and associate attributes, including service provider, IP address and service description, etc. Service providers can add additional attribute names and values when the need arises. It provides a high-level description of services, which typically would be presented to users when browsing a service registry. It will be used during the attribute matching.

For semantic description, services are represented by DAML descriptions, which consist of ontology and instances, in terms of their capabilities and functionalities. The ontology defines a set of classes and properties specifically for describing services, which consists of three types of information: a human readable description of the service, a specification of the functionalities that are provided by the service, and a list of functional attributes which provide additional information and requirements about the service that assist when reasoning about several services with similar capabilities. The functional specification of a service is represented in terms of inputs, outputs and preconditions. An example of advertisement is shown in Fig. 3. It shows a MP3 music service returns which music can be brought to a requester when presented with a music title and a credit card number.

```
<profile:Profile rdf:ID="MP3MusicService">
  <profile:serviceName>mp3musicService</profile:serviceName>
  ......
  <profile:input>
     <profile:ParameterDescription rdf:ID="MusicTitle">
        <profile:parameterName>musicTitle</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://www.w3.org/2000/10/XMLSchema.xsd#string" />
        <profile:refersTo rdf:resource="http://lucan.ddns.comp.nus.edu.sg/octopus/daml/MP3.daml#musicTitle"/>
     </profile:ParameterDescription>
  </profile:input>
  <profile:input>
     <profile:ParameterDescription rdf:ID="CreditCardNumber">
        <profile:parameterName>creditCardNumber</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://www.w3.org/2000/10/XMLSchema.xsd#decimal" />
        <profile:refersTo rdf:resource="http://lucan.ddns.comp.nus.edu.sg/octopus/daml/MP3.daml#creditCardNumber
     </profile:ParameterDescription>
  </profile:input>
  <profile:ouput>
     <profile:ParameterDescription rdf:ID="MusicOuput">
        <profile:parameterName>music</profile:parameterName>
        <profile:refersTo rdf:resource="http://lucan.ddns.comp.nus.edu.sg/octopus/daml/MP3.daml#Music"/>
     </profile:ParameterDescription>
  </profile:ouput>
  ......
</profile:Profile>
```

Fig. 3. Advertisement of a service in DAML+OIL.

A service can also be described and registered using Jini's interface as specified in (Waldo and Arnold, 2000).

A SLS client uses Java RMI to connect to a SLS server providing coverage for its area, and submit a query in any form mentioned above. The SLS client can also specify in the query to allow the system to convert its query to other forms to invoke corresponding matching mechanisms. Converting between different forms is performed by the Service Locating Manager in a SLS server. Constraints can be specified in the query, for example, the SLS client may specify to return the result immediately once a service is found to achieve the best response time.

4.4. SLS servers

A SLS server consists of Service Locating Manager, Service Aggregator and SLS server manager as shown in Fig. 4. Their functionalities are described.

4.4.1. SLS server manager

The SLS server manager is responsible for server management. Its functionality includes server self-configuration, building and updating the dynamic server tree and keeping track of server relationship. It creates and updates the following parameters in a server.

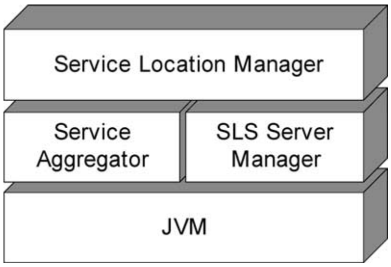

Fig. 4. Key components of a SLS server.

- Server name. It contains the name of a SLS server.
- Server GUID (Global Unique Identifier). It is used to identify each SLS server. Each server has one unique GUID.
- Server level. It designates the level of a server.
- Parent/child server. It keeps a reference to its parent server and its child server.

4.4.2. Service aggregator

The Service Aggregator is an entity which analyses service information from other SLS servers and aggregates them into abstracted service information in a compact form. Its functionality is to perform service aggregation when creating, updating and deleting of SLS servers or services. Service aggregation is done by abstracting service type information from a wide range of mobile services. For example, m-commerce applications, which are represented by the root service-*mServiceAll*, can be categorized into main categories: Mobile Information Services-*mInformation*, Mobile Directory Services-*mDirectory*, Mobile Banking Services-*mBanking*, Mobile Entertainment-*mEntertainment*, etc. Main categories and sub-categories can be further divided. For example, *mEntertainment* can be divided into: *Video*, *Music*, etc.; *Music* can be divided into *MP3*, etc. When a new service registers itself into a server, its abstracted service information gets updated in the local server and will be propagated to its parent servers until reaching the root server as illustrated in Fig. 5.

In the case of attribute-based service representation, a Service Type Table is used to store abstracted service information. The Service Type Table keeps all abstracted service information that a SLS server can support. For services represented in Jini's interface, the same aggregation scheme is followed but with a different set of Service Type tables.

In the case of semantic service representation, service aggregation is achieved by profile-based class hierarchy and managed by Class Hierarchy Table. The Class Hierarchy Table keeps a set of hierarchy subclasses through which a SLS server can provide services. We construct a hierarchy of subclasses of the *mProfile* class to categorize a broad array of mobile services. Each subclass inherits the properties from its superclass. Each SLS server contains a set of class hierarchy in which it can provide these services; and its parent server stores the superclass. In this way, service instances can be distributed among SLS servers where a parent–child relationship exists based on the class hierarchy. Each SLS server maintains a Class Hierarchy Table indicates which services the SLS server can

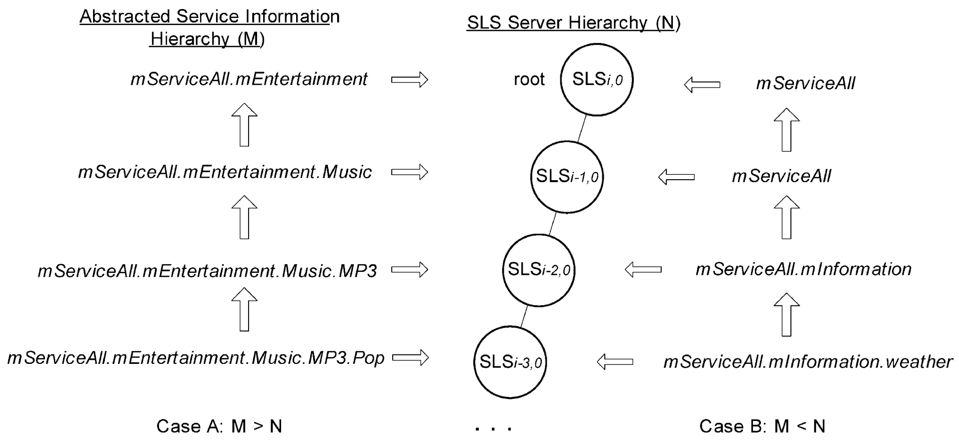
Abstracted Service Information Hierarchy (M)
SLS Server Hierarchy (N)
mServiceAll.mEntertainment
mServiceAll.mEntertainment.Music
mServiceAll.mEntertainment.Music.MP3
mServiceAll.mEntertainment.Music.MP3.Pop
root
SLSi,0
SLSi-1,0
SLSi-2,0
SLSi-3,0
mServiceAll
mServiceAll
mServiceAll.mInformation
mServiceAll.mInformation.weather
Case A: M > N
. . .
Case B: M < N

Fig. 5. Service aggregation by class hierarchy.

support. If there are any new DAML ontologies and instances registered or there is a structural change on its children servers, the Class Hierarchy Table will be updated.

4.4.3. Service locating manager

The functionalities of the Service Locating Manger include converting different query forms, service registration, service matching, and consistency check for service ontologies and instances. Query conversion is done in a SLS server; hence it can offload the burden of processing in a resource-constrained mobile device and minimize the message transferred from client to server. The Service Locating Manger checks the consistency of each service or ontology registered. Services with different forms are stored in different databases. Service matching is done by multiple service matching engines, which consist of an attribute matching engine and a semantic matching engine. We also retain the interface matching employed by Jini. In Jini architecture, a service is registered as a Java/Jini object to LUS. The Jini interface matching engine in LUS maps interfaces indicating the functionality provided by a service to sets of objects. The corresponding matching engine will be automatically invoked depending on the form of a query.

Attribute matching engine. In the attribute matching, both services and queries are defined using service templates. The attribute matching mechanism is based on a frame-based search engine, which increases its precision compared to keyword-based search engines at the cost of requiring that all services be modelled as frames using templates. It will match services whose service type and attribute values equal to those in the query. The results for the search will be returned to the client in a ranked order if multiple service instances have been found. The ranking is determined by the number of matched attributes and the priority of attributes.

Semantic matching engine. The architecture of the semantic matching engine is shown in Fig. 6. For semantic matching, services are marked up in DAML+OIL. A service provider registers its DAML service ontologies and instances with any SLS server. The service ontological information and instances are presented as inputs to the DAML reasoner. The reasoner parses each statement in the ontologies and instances; checks

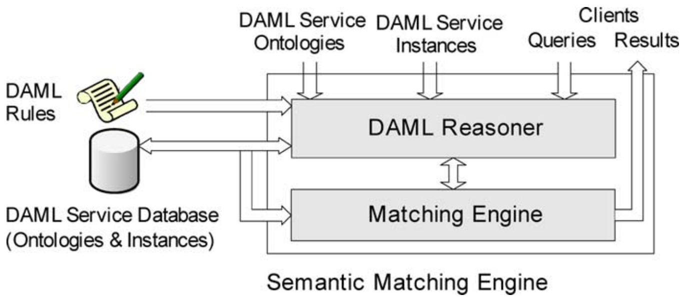

Fig. 6. The architecture of the semantic matching engine.

the validity of each statement to ensure they conform to the ontology. Then the reasoner loads the ontology, the instances and relationship rules into its Knowledge Base. When a SLS client makes a query through the application GUI, the query is converted to a DAML description and parsed by the reasoner for checking of validity. If the checking succeeds, the reasoner will parse all DAML statements to RDF triples (Resource Description Framework) and perform semantic matching. The results will return to the client when a match is found.

5. Implementation and performance

In this section, we present our implementation and performance evaluation. We implemented the SLS system based on Java J2SE 1.3.1. We tailored the J2SE 1.3.1 RMI runtime classes to accompany the need of resource-constrained mobile devices.

A testbed for performance evaluation has also been built as shown in Fig. 1. Each SLS server ran in a 1.6G Intel Pentium 4 machine with 256 MB RAM. A SLS server needs to configure itself before its startup. The configuration includes specifying server name, server level, unicast address, and downloading service type information from a global service type repository and common ontologies from a service ontologies repository. We assume that all the available service types and common ontologies are stored in the global repository, which is located somewhere in the network. In our experiments, we set the default lifetime to 10 s for multicast and 2 s for unicast. All SLS servers can be discovered within the time limit. The average latency between a pair of parent–child servers is about 1.2 ms. We also tested the dynamic tree by adding one or more additional SLS servers to the existing tree in the testbed; and removing one or more SLS servers from the existing tree. Server joins and leaves the tree smoothly; and the tree adapts to the changes quickly. For example, the system takes 4.2 ms to update the server tree when a new SLS Server F wants to join C.

To measure the overhead of converting between different query forms, a SLS client sent out a query using a specific form, i.e. a service template, with a request of conversion. A SLS server received the query and convert to other forms. The worse case of the overhead occurred when converting a service template to a DAML description; takes about 0.5 ms. The overhead for the rest of converting cases are less than 0.2 ms.

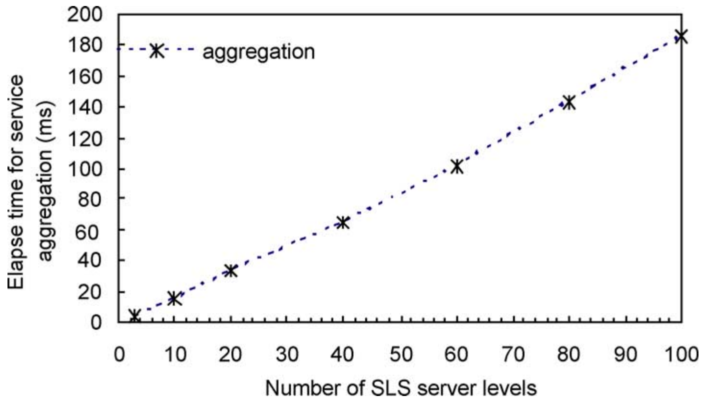

Fig. 7. Performance of service aggregation.

5.1. *Performance of service aggregation*

The performance of our service aggregation scheme depends on number of service types and number of subclasses registered. The runtime of checking and updating a service type or a subclass is less than 1 μs. We studied how the number of server levels affects the performance of service aggregation by running a simulation. We limited the service type hierarchy or class hierarchy to 50; and set up a different number of SLS server levels with a child-to-parent relationship. Each time, a new service registered into the lowest server and aggregation is performed upward until reaching the root server. We measured the elapse time for service aggregation as shown in Fig. 7. The result shows our service aggregation scheme is linear in the number of server levels or scalable to number of servers.

5.2. *Searching performance*

We measured the system searching performance in two experiments. In Experiment 1, with 400 attribute-based service instances in each SLS server in the testbed, multiple mobile clients emulated using Pentium laptop with wireless LAN connectivity on separate machines made concurrent requests to different servers. We measured the average elapse time taken by our SLS system for different number of concurrent requests as shown in Fig. 8. For each experiment, the measurements were repeated for 10 times and the average results were collected. We found that the average elapse time for each search request is around 250 ms. The average elapse time for attribute searching is nearly proportional to number of concurrent requests. The attribute matching mechanism performs well as number of requests increases. It also demonstrates the SLS system is reasonably scalable with respect to number of users.

In Experiment 2, a SLS client on a separate machine made 10,000 requests continuously to Server D. We tested the attribute matching performance for different numbers of service instances running in each SLS server. Each time, a SLS server stored different number of attributed-based service instances starting from 100 to 600 instances.

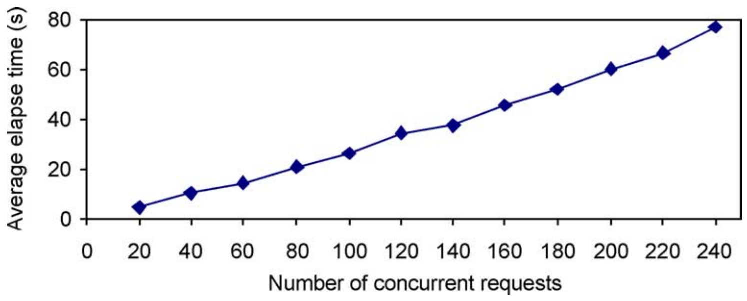

Fig. 8. Average elaspe time of concurrent requests for 400 service instances in each SLS server.

We measured the average elapse time per request for different number of instances as shown in Fig. 9.

The elapse time for attribute matching is also nearly proportional to number of service instances. When the SLS servers store more service instances, the attribute matching performance will gradually decrease. This can be explained below. In our current implementation, all services are stored in memory of SLS server machines. When number of services, the required memory also increases. The operating system is increasingly spending more time in memory swapping, which deteriorates performance. Adding more memory on the SLS server machines will help to improve the performance. In the future, we plan to store service instances into database that can deal with large volume of data and hopefully this can better the searching performance.

5.3. Performance of the multiple matching engines

We implemented the attribute matching engine and the semantic matching engine. For the semantic matching, we adopt JTP (Fikes et al., 2003) as our DAML reasoner as it is an object-oriented modular reasoning system and it is easy to add-in a user specific reasoning modular. A query is specified by a DAML description through the application GUI and then converted to Knowledge Interchange Format which JTP can recognize.

We measured and compared the performance of attribute matching, semantic matching engines and Jini's interface matching. With 400 service instances stored in

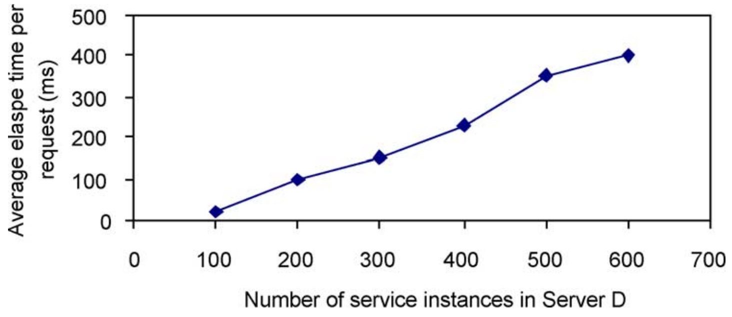

Fig. 9. Average elaspe time of each search request for different number of service instances.

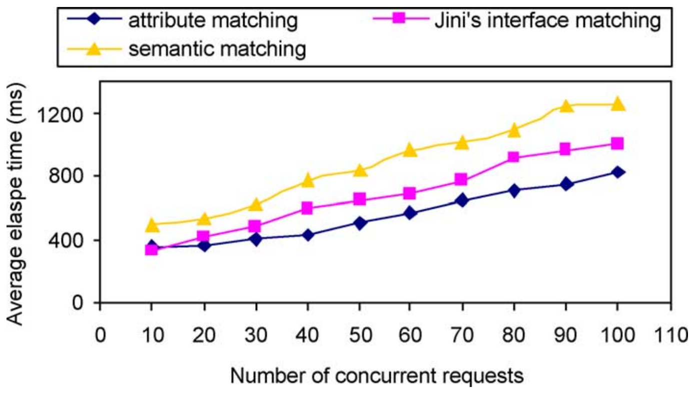

Fig. 10. Performance comparison of multiple matching engines.

Server D using different forms, a SLS client on a separate machine made multiple concurrent requests to Server D. We measured the average runtime per request for the three matching engines. For each matching engine, the measurements were repeated for 10 times and the average results were collected. As shown in Fig. 10, the attribute matching and the Java interface matching performs better than the semantic matching. They have a similar trend that is the average runtime per request increases proportionally to the number of concurrent requests. By further studying the factors affecting the average runtime of a search request for the semantic matching engine, we found the reasoning process has consumed a significant time in the overall matching process. It is the major factor which caused the performance difference compared with the other two engines.

We also find that the runtime for RMI operation is another important factor in terms of overall performance. As Java RMI is used in our system as a basic communication mechanism between a client and a server, hence, enhancing Java RMI operation will significantly improve the performance of all the three matching engines.

6. Conclusion

We have proposed and implemented a prototype of a flexible service discovery system known as SLS for m-commerce applications. The system automatically adapts its behaviour to handle dynamic changes of both SLS servers and services, hiding the complexities of internal mechanisms from users and service providers. The system is dynamic as it adopts the dynamic tree structure, and is flexible as it has multiple service matching mechanisms. The idea of service aggregation is embedded in our system, which is used for a faster searching and minimizing communication cost. In our future work, we will test the scalability of our prototype system in a wide-area network setting.

References

Adjie-Winoto W, Schwartz E, Balakrishnan H, Lilley J. The design and implementation of an intentional naming system Proceedings of ACM Symposium on Operating Systems Principles 1999 p. 186–201.

Ankolenkar A, Burstein M, Hobbs JR, Lassila O, Martin DL, McDermott D, McIlraith SA, Narayanan S, Paolucci M. DAML-S: web service description for the semantic web Proceedings of the First International Semantic Web Conference (ISWC) 2002.

Balazinska M, Balakrishnan H, Karger D. INS/Twine: a scalable peer-to-peer architecture for intentional resource discovery Proceedings of the First International Conference on Pervasive Computing, Zurich, Switzerland, August 2002 p. 195–210.

Bellwood TA. UDDI—a foundation for web services Proceedings of XML Conference and Exposition. Orlando, Florida, December 2001.

Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana S. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput 2002;.

Czerwinski SE, Zhao BY, Hodes T, Joseph AD, Katz R. An architecture for a secure service discovery service Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks, Seattle, WA, August 1999.

Enabling UMTS/third generation services and applications. UMTS Forum Report 11, October; 2000.

Fikes R, Jenkins J, Frank G. JTP: a system architecture and component library for hybrid reasoning Proceedings of the Seventh World Multiconference on Systemics, Cybemetics and Informatics, Orlando, Florida, USA, July 2003.

Gu T, Qian HC, Yao JK, Pung HK. An architecture for flexible service discovery in OCTOPUS Proceedings of the 12th International Conference on Computer Communications and Networks (ICCCN), Dallas, Texas, October 2003.

Horrocks I. DAML+OIL: a reason-able web ontology language Proceedings of the Conference on Extending Database Technology (EDBT 2002) March 2002.

Java remote method invocation—distributed computing for Java. http://java.sun.com/marketing/collateral/javarmi.html

Resource Description Framework. World Wide Web Consortium. http://www.w3c.org/rdf

Sadeh N. m-Commerce technologies, services and business models. New York: Wiley; 2002.

Simple Object Access Protocol (SOAP). W3C. http://www.w3.org/TR/SOAP

The Salutation Consortium. Salutation architecture specification (part 1) version 2.1 edition 1999 http://www.salutation.org.

UPnP White Paper. June; 2000. http://upnp.org/resources.htm/.

Waldo J, Arnold K. The jini specification second edition, 2nd ed. Reading, MA: Addison-Wesley; 2000.

Web Services Description Language (WSDL). W3C. http://www.w3c.org/TR/wsdl12

Tao Gu (http://www.comp.nus.edu.sg/~gutao) is a PhD candidate in the School of Computing at the National University of Singapore. His current research interests include pervasive computing, context-aware systems, peer-to-peer systems and service discovery. He received his MS in Electrical and Electronics Engineering from Nanyang Technological University, Singapore. He is a member of IEEE. Contact him at gutao@comp.nus.edu.sg.

Dr Hung Keng Pung (http://www.comp.nus.edu.sg/~punghk) received his BSc in Communications Engineering and his PhD in Electronics from the University of Kent at Canterbury, UK, in 1981 and 1985, respectively. He is an associate Professor of the Department of Computer Science, National University of Singapore. He heads the Network Systems and Services Laboratory and a faculty associate of the Institute of Infocomm Research (I2R) in Singapore. His main research interest is on adaptive network systems, protocols design and networking, quality of service, and mobile commerce middleware.