IDEA GROUP PUBLISHING

ITP4124

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

Intelligent Service Discovery

Robert Bram and Jana Dospisil
School of Network Computing, Monash University, Australia, {Rob.Bram, Jana.Dospisil}@infotech.monash.edu.au

ABSTRACT

The claim of improved efficiency and reliability of networking technology provides for a framework of service discovery where clients connect to services over the network based on a comparison of the client's requirements with the advertised capabilities of those services. Basic service discovery in Jini involves an exact pattern match between attributes in the clients requirement description and the services' capability description. Advanced service discovery should allow for comparative pattern matching using attributes that measure aspects of a service and offer the client a far more refined power to choose the best services for them. This paper proposes a framework of intelligent service discovery using a generic constraint satisfaction problem solving architecture that uses comparative pattern matching and allows search algorithms to be used as library components.

INTRODUCTION

The availability, speed and reliability of networking technology validates a service discovery framework where clients connect to services over the network based on the advertised capability descriptions [9] of those services. Services in this framework are interfaces to devices, applications, objects or resources that the client needs.

The key challenge for such systems is to enable clients to locate the service that best suits their needs, where 'best' will be defined by the client [10] using infrastructure provided by the framework's implementation. Example definitions may include type of service or quality of service such as cost, speed or accuracy of results. The client's requirements must be compared with the advertised capability descriptions of the services to find the best match [9].

Service discovery frameworks may be classified according to locality (where the comparison takes place): at the client site, at the server site, or a third party search service called a *lookup*. *Lookup* is typically a directory based process of locating (looking up) a specific service or activating an agent capable of doing the job [7]. Each of these styles has a different profile in terms of network traffic and a combination of all three is possible [9].

When using a lookup, the client must be able to provide the lookup process with enough detail for the service to be located. This detail may be a specific address or identification or it may be data to form some matching criteria with which the lookup process may search upon and build a satisfier set [5] from which a service may be selected.

This paper shall begin by outlining how a directory enables service discovery and how Jini's service discovery mechanism utilises exact pattern matching

This paper begins by showing the basic definition and usage patterns of a directory as they apply to any lookup discovery service and to Sun's Jini technology in particular. It shall provide justification for an intelligent search service using comparative pattern matching and propose a framework that treats a search as a constraint satisfaction problem.

INTRODUCING THE DIRECTORY

A directory is any searchable list of references: usually ordered and thus indexed. Directories may be subdivided into smaller lists or categories and may even be indexed according to these categories.

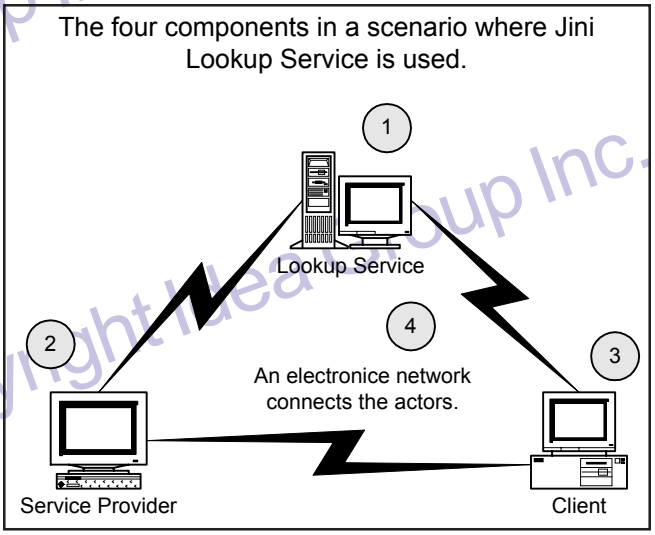
The service providers who wish to "sell" their services must register the services with the directory. The client searches available directories to find a set of suitable services by comparing their requirements with details about each service provided by the directory. The references registered with the service provides pointer to the service or a proxy.

Jini: A Java Directory

In a peer-to-peer environment, directories are central repositories for information. Directory information software can be scalable, granular and independent of location and able to present and store meta-data [2].

Jini is a set of Java API's defining an electronic directory service designed to list references to other electronic services. There are four main components to Jini: three actors connected via a network as illustrated in Figure 1 below.

Figure 1: Components in a Jini Scenario



In Jini terminology, the directory is a *lookup service* (or service locator or service registry). The references are proxy objects that have been registered with the lookup service by the service providers that created them. The searchers are clients who conduct searches on the lookup for a set of service objects matching a service template they created.

The same general patterns are used in Jini to dictate how references make their way into the lists and how searchers use the directory.

1. Service providers discover the *lookup service*,
2. service providers register a service item with the lookup service that include a service proxy that represents the actual service to the client and a set of *Entry* attributes that describe the service in various ways.

The same pattern defines how clients (searchers) make use of a *lookup service*:

1. clients discover the *lookup service*;
2. clients lookup the service(s) they desire by sending a service template that represents search matching criteria. The service template can include a service id to reference a specific service or it may include an array of object types that the service object must imple-

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference.

ment or it may include an array of Entry objects that represent desired state to be searched for;
3. the lookup service returns a set of proxy objects that satisfy the search criteria;
4. clients choose which proxy to use, which may or may not involve further interaction with the service provider.

Exact Pattern Matching

Jini provides a lookup service (Reggie) that allows a client to use any of three search criteria:

- a *ServiceID*, guaranteed to be globally unique for each Jini registered service;
- a set of *Class* objects - object types matching the required service or interfaces implemented by the required service; and
- a set of *Entry* attributes describing the service.

When the client submits a *ServiceTemplate* to the lookup service, a search is performed on all three criteria: an exact pattern match will be performed on the Entry attributes of the *ServiceItem*s in the satisfier set formed by the first two criteria.

An attribute in the *ServiceTemplate* is matched by an attribute in the *ServiceItem* if all fields of the attribute that are non null match exactly their corresponding field in the *ServiceItem*. This is the exact pattern match implemented by Jini [4]. It is important to remember that the attributes are kept serialized even for comparison – the value of two fields are considered matched if have the same sequence of bytes according to the Java serialisation scheme [4]. :

This form of pattern matching keeps the mechanism quite simple. It also means that code need not be deserialised to allow for custom compare functions [14]. If a client wishes to conduct more advanced searching, they need to define a *ServiceTemplate* that will match the broadest category of services they are interested in and implement an advanced search on that set.

This scenario does not represent a good separation of concerns: clients should be free to perform the tasks they are assigned without having to manage specialised service discovery code that could involve a great deal of calculation to find the correct service, such as a client searching for the Solution Engine for a Constraint Satisfaction Problem [12]. This specialised code belongs in a service of its own, for use by clients in need of the same.

PROPOSED INTELLIGENT SEARCH SERVICE FRAMEWORK

An intelligent search service should be able to conduct a comparative search upon the terms handed to it. For example, the service could be designed so as to accept a predicate logic expression [5] or predicate object [6] that could be evaluated to form a satisfier set [5] for the client to choose from.

When a lookup service is given a *ServiceTemplate*, it attempts service discovery on behalf of the client by searching a domain (the directory) for a solution (the satisfier set). This suggests that searching for a service could be described as a constraint satisfaction problem. A constraint satisfaction problem (CSP) is any problem that can [8]:

a) be defined by:
   - a set of variables: $V = \{v_1, v_2, \ldots v_m\}$,
   - a set of domains that define what values each variable can take: $D = \{d_1, d_2, \ldots\ldots, d_m\}$,
   - a set of constraints that define all relationships between the variables: $C = \{c_1, c_2, \ldots\ldots, c_n\}$;
b) solved by determining a set of variable-value pairs that satisfy all constraints (a *solution*).

Finding a solution to a CSP means finding a set of value assignments for each variable such that all constraints are satisfied. The basic process is iterative. First select a variable for instantiation, then select a value and assign it to the variable and determine whether the assignment is consistent with all of the constraints. If an inconsistency is detected, backtrack; otherwise iterate with the next variable: this step is controlled by the heuristics of the algorithm, deciding, for example, what variable is to be considered next and in what order the set of constraints should be evaluated.

From the above broad description of a constraint satisfaction problem, it can be seen that any search implemented as a CSP must be able to model the following elements:

- variables
- domains
- constraints
- heuristics

All search criteria of a Jini *ServiceTemplate* are variables – they are object data members of the *ServiceTemplate*. Java objects or primitives have a primary domain automatically specified by their type. This suggests that a collection of java objects (and primitives) are by default a set of matched variables and domains.

Discounting primitives, variables are Java objects that maintain state and may define a more intricate non-contiguous domain via the inclusion of a method that outputs the next value in the domain's sequence.

Constraints model relationships between two or more variables. "Cost <= budget" is a simple example that illustrates a simple point: a constraint, in essence, is a logic expression that evaluates to true or false. If variables are modelled as Java objects, a constraint's logic could be implemented by a method that returns a boolean value and has access to all variables included in the relationship.

Earlier it was stated that heuristics should be able to control the order in which constraints are evaluated. If each individual constraint is modelled as an individual object, the set of all constraints may be stored in an array and ordered to suit.

The processing of a set of heuristics in between each iteration of a constraint problem solving exercise involves modifying the order in which variables are assigned values and constraints are evaluated. The collections of variables and constraints can be stored in their own arrays, made available to a Java object encapsulating heuristic logic which can order the variables and constraints as it sees fit, read for the next algorithmic iteration.

Intelligent Service Discovery Attributes

A set of intelligent service discovery attributes may be devised to model variables, domains, constraints and heuristics as Entry objects that can be included in a standard Jini *ServiceTemplate*.

```
interface ISDVariable implements net.jini.core.Entry
{
  public Object variable;
  public String variableName;
  public Object nextValue ();
}

interface ISDConstraint implements net.jini.core.Entry
{
  public boolean checkConstraint (ISDVariable [ ] attributes);
}

interface ISDHeuristics implements net.jini.core.Entry
{
  public void processHeuristics
      (ISDVariable [ ] attributes, ISDConstraint [ ] constraints);
}
```

Generic Service Discovery Algorithm as a CSP

Objects implementing the interfaces from section 3.1 will contain the logic required to find the best set of services where best is

defined by the client. The task is modelled as a constraint satisfaction problem and the generic iterative algorithm below shows a solver acting as a compute engine [15].

```
BEGIN main
  DO
      ISDHeuristics.processHeuristics (attributes, constraints)
     attributes [n] = attributes [n].nextValue ()
  UNTIL constraintsSatisfied
END main
BEGIN constraintsSatisfied
   boolean result = true
   FOR all ISDConstraint in constraints
      result = result AND constraints [n].checkConstriant (attributes)
  END FOR
END constraintsSatisfied
```

Different constraint solving algorithms can be handed to the intelligent search service as different configurations of *ISDVariable*, *ISDConstraint* and *ISDHeuristics* objects , allowing the client to specify what strategies or heuristics they wish to have the search [12**Error! Reference source not found.**].

When a client is ready to perform lookup on a Jini ServiceRegistrar (the lookup service's proxy), they may specify the maximum number of matches to be received. It should also be possible for

Intelligent Search Service as a new Service

Jini defines a small set of standard attributes in net.*jini.entry.AbstractEntry*: *Address*, *Comment*, *Location*, *Name*, *ServiceInfo*, *ServiceType* or *Status*. Together, these attributes form the primary model of Jini's exact pattern matching during lookup and effectively describe a style of attributes that act as key words, much as the html meta keyword tag.

A lookup that services complex and simple searches could end up slowing delivery time for the simple searches. Instead, the two services can exist side by side. A client who needs a simple search can use the Jini lookup service. A client who needs an intelligent search can lookup the intelligent search service and then use it, saving the Jini lookup from the extra work.

Intelligent Attributes Measure and Describe

Attributes of a service are by definition *descriptive*. The standard set of attributes mentioned in section 0 contain descriptive string information about a service. Comparative service discovery should uses descriptive *measures* of a service: any measurable quality can be compared with the inequality operators, < and > as well as the equality operator, =.

Potential candidates for intelligent attributes include:

- size of the service proxy to be downloaded;
- bandwidth of the service provider;
- expected minimum, maximum and median processing time needed to run service (or maybe a formula for the calculation of expected processing time);
- queue length for service (see NOTE below);
- cost of the service.

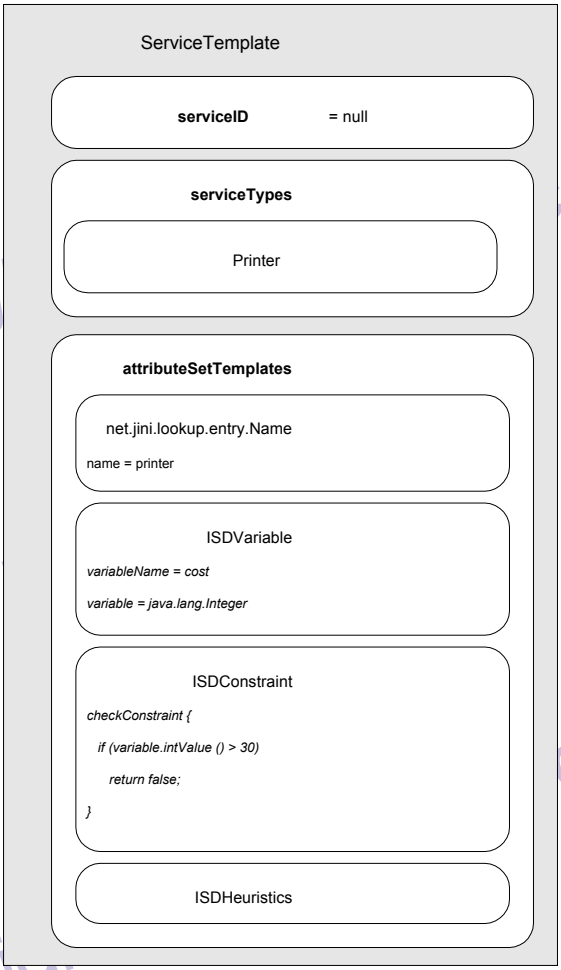
NOTE: using an intelligent service discovery service will undoubtedly take longer than ordinary lookup. If attributes are used to measure highly dynamic values of a service it is important to note that the value may change by the time the client gets the results.

ServiceTemplate for Simple and Complex Searches

The search service will a receive a ServiceTemplate and discriminate between 'simple' and 'complex' attributes based on their type. A sample ServiceTemplate is shown in Figure 2.

The intelligent search service would perform a standard lookup to receive a set of services that match the 'simple' attributes. It shall then search that set for the client's satisfier set that meet the criteria specified by the *ISDVariable*, *ISDConstraint* and *ISDHeuristics* objects.

Figure 2: Example search object



A Generic Framework of Service Discovery

Jini is being used as a test bed for this idea. The application will provide a search service for other jini services, but there is no reason why it cannot be designed to facilitate a client connecting directly to the search service without knowledge of Jini.

The most important aspect is to ensure that a communication protocol is in place – using either XML or Java objects to contain the service template and results.. Wrappers could be written for RMI or SOAP, with the wrapping class incorporating the appropriate transmission protocol as well – Sockets etc.

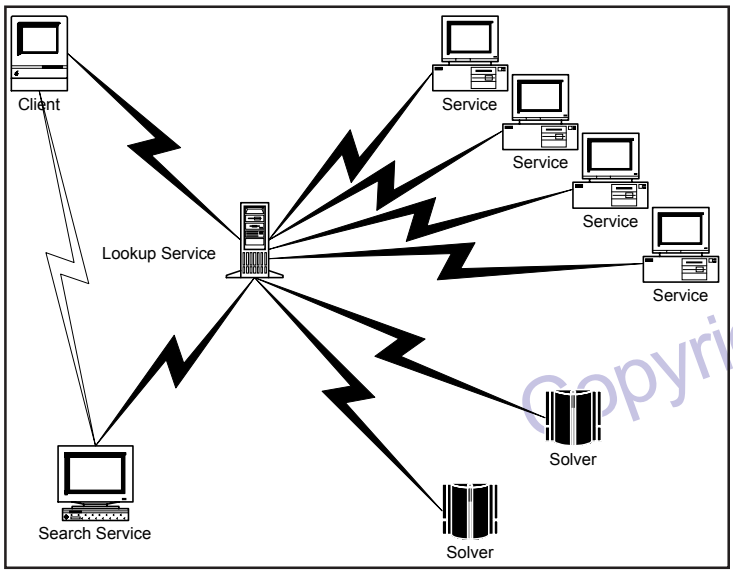
A client will need to discover the lookup service and then lookup the search service. The search service proxy would be a remote object and clients call lookup on it, sending a new search object and receive the results object containing references to the services found (proxy objects the search service downloaded from the lookup service).

CONCLUSION AND FUTURE WORK

This paper represents the current state of research still in progress. A framework of intelligent service discovery has been outlined using a generic constraint satisfaction problem solving architecture that allows different algorithms to be used as library components.

The addition of an intelligent search service would undoubtedly increase the complexity of an already complex notion. The benefits

Figure 3: Service discovery with a search service



of such a search service are that clients in a large system would have the power to select the service that is right for them based on attributes that describe measurable aspects of the service rather than just describe the service (i.e. the basic Jini attributes act more as keywords for a service). This power to discriminate is useful in any scenario where there is a large number of services with similar attributes. An example is a market place driven by competition for client patronage or a pool of Solvers that all solve Constraint Satisfaction Problems but with different, specialist, algorithms.

The client must be sure such a service is worth their while; a search service would undoubtedly increase the time it takes to link the client to a service, so it must be proportionately important to the client to find the right service.

In addition to increasing the client's time to connect to a search service, there is the possibility that bottlenecks will hamper the performance of any central directory service However, an intelligent search service splits the work of service discovery into two tiers: simple discovery and complex discovery. The search service, being specialized, will not receive lookup requests that Jini's native lookup service can handle.

Another load balancing issue has to do with dynamic attributes and the fact that service discovery is not an atomic action. The time delay in using a search service could mean that a critical variable used to make a decision based on its current value could have quite a different value when the client finally connects to the service. This warning applies to any dynamic attribute. One method of dealing with dynamic or volatile attributes (whose values change often) is to include a field rating the volatility of an attribute. Highly volatile attributes might be considered with a lower priority than non volatile attributes. These functions can be relegated to the reasoning implemented by a ISDHeuristics object.

Backtracking and constraint propagation are the two main categories of algorithms used to solve constraint satisfaction problems [8]. For testing purposes a highly simplistic algorithm will be implemented, but an important question that needs to be answered by this investigation is: will the framework outlined in this paper be capable of allowing a wide variety of established CSP algorithms to run?

Future work will involve completing the test bed and developing a suite of standard *ISDVariable*, *ISDConstraint* and *ISDHeuristics* classes to allow a variety of search algorithms to be used 'out of the box' – without the client having to get their hands dirty except for filling in a few values. The intelligent search service will be evaluated according to how successfully it discovers the best services for a client as defined by the collection of *ISDConstraint* and *ISDHeuristics* objects.

REFERENCES

Definition of HAVi appears here:http://whatis.techtarget.com/definition/0,,sid9_gci748239,00.htmlURL last accessed: 29 September 2001.More information can be found here:www.havi.orgURL last accessed Tuesday 2 October 2001.

Directory Services Markup Language by Bowstreethttp://www.dsml.org/about.htmlURL last accessed Wednesday 3 October 2001.

Edwards, K. W. Core Jini, Second Edition. Prentice Hall, PTR, Upper Saddle River, 2001. Page 270

Edwards, K. W. Core Jini, Second Edition. Prentice Hall, PTR, Upper Saddle River, 2001. Page 248.

Hughes E., Mc Cormack D., Barbeau M. and Bordeleau F. Service Recommendation using SLP (Service Location Protocol). Written for the IEEE International Conference on Telecommunications (ICT), Bucharest, June 2001.http://www.scs.carleton.ca/~barbeau/Publications/2001/ICT/hughesmccormack.pdf. URL last accessed Monday, 1 October 2001.

IBM – SLAPHAPIhttp://www.dsml.org/dsml_in_action/ibm.htmlURL last accessed Wednesday 3 October 2001.IBM - SLAPHAPI (Standalone LDAP HTTP API) is a generalized high-level application programming interface for accessing LDAP directories via HTTP requests – part of the directory services markup language (see [2]).

McGrath E. R. Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing. Presented at Center for Excellence in Space Data and Information Science, NASA Goddard Space Flight Center. April 5, 2000.http://www.ncsa.uiuc.edu/People/mcgrath/Discovery/dp.htmlURL last accessed: Monday, 1 October 2001

P. Crescenzi, G. Rossi. On the Hamming Distance of Constraint Satisfaction Problems. Dipartimento di Sistemi ed Informatica, Università di Firenze. 25 January 2000. http://gdn.dsi.unifi.it/~rossig/Papers/TCS1/tcs.html URL last accessed Wednesday 3 October 2001.

Pascoe R. Dynamic networking requires comprehensive service discovery. Serverworld's HP Chronicle Archives: October 2000 Issue.http://www.serverworldmagazine.com/hpchronicle/2000/10/discovery.shtml.URL last accessed Monday 1 October, 2001.

Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. Secure Service Discovery Service: An Architecture for a Secure Service Discovery Service. For Mobicom '999 Seattle, Washington USA.http://iceberg.cs.berkeley.edu/papers/Czerwin-Mobicom99/sds-mobicom.pdfURL last accessed: Monday, 1 October 2001

Sun Java site for downloading the Jini API's.http://www.sun.com/software/communitysource/jini/download.htmlURL last accessed Wednesday 3 October 2001.

Tsang E., Kwan A. Mapping Constraint Satisfaction Problems to Algorithms and Heuristics. For the Department of Computer Science, University of Essex. December 15, 1993.

Venners B. Objects, the Network, and Jini. First published under the name Jini: New Technology for a Networked World in JavaWorld, a division of Web Publishing, Inc., June 1999.http://www.artima.com/jini/jiniology/intro.htmlLast accessed Monday October 1, 2001.

Venners, B. Finding Services with the Jini Lookup Service - Discover the power and limitations of the ServiceRegistrar interface. First published under the name "Finding services with the Jini lookup service" in JavaWorld, a division of Web Publishing, Inc., February 2000.http://www.javaworld.com/javaworld/jw-02-2000/jw-02-jiniology.htmlURL last accessed Tuesday 2 October 2001.

Sun's Java tutorial for RMI outlines a Compute Engine.http://java.sun.com/docs/books/tutorial/rmi/URL last accessed Thursday 4 October 2001.