Lookup

Michael Kircher & Prashant Jain

{Michael.Kircher,Prashant.Jain}@mchp.siemens.de

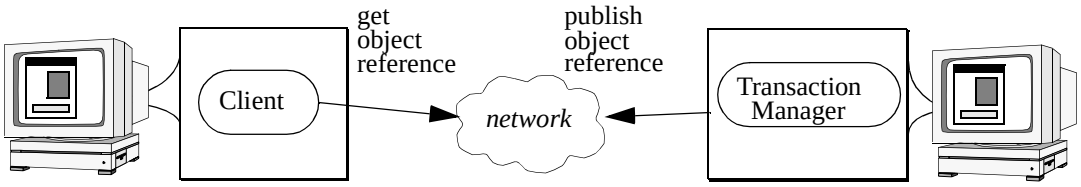
Siemens AG, Corporate Technology

Munich, Germany

Lookup

The lookup pattern describes how to find and retrieve initial references to distributed objects and services.

Example Consider a system consisting of several distributed objects implemented using CORBA. To access one of the distributed objects, a client typically needs to obtain a *reference* to the object. An object reference identifies the distributed object that will receive the request. Object references can be passed around in the system as parameters to operations as well as results of requests. A client can therefore obtain a reference to a distributed object in the system from another distributed object. However, how can a client get an *initial* reference to an object assuming that it is the first distributed object the client wants to access?



For example, in a system providing distributed transaction service, a client may want to obtain a reference to the Transaction Manager to be able to participate in distributed transactions. How can a server make the Transaction Manager object reference that it created widely available? And how can a client obtain the Transaction Manager object reference without having a reference to any other object?

Context Distributed systems where clients need to retrieve initial references to distributed objects or services.

Problem In a distributed system, a server may offer one or more services to clients. Over a period of time, additional services may get added or existing services may get removed. One way the server can publish the availability of existing services to interested clients is by periodically sending a broadcast message. The messages need to be sent on a periodic basis to ensure that new clients that join the system become aware of available services. Conversely, a client could send a broadcast message requesting all available services to respond. Once the client receives replies from all available services, it can then choose the service(s) it needs. However, both these approaches can be quite costly and inefficient since they proliferate the network with lots of messages. To address this problem of allowing servers to publish services and for clients to find these services in an efficient and inexpensive manner requires the resolution of the following *forces*:

- *Availability*: a client should be able to find out on demand what distributed objects or services are available in its environment.
- *Independence*: a client should be able to obtain initial references of one or more distributed objects or services without relying on other unknown distributed objects or services.
- *Location Transparency*: a client should be able to obtain initial references of one or more distributed objects or services without caring about the location of the distributed objects or services. Similarly, a server should be able to provide references of distributed objects and services to clients without knowledge of the location of the clients.

- *Simplicity*: the solution should not burden a client obtaining the initial references of distributed objects and services nor a server providing the references.

Solution Provide a Lookup service which allows services to register their references and clients to retrieve these references. The Lookup service serves as a central point of communication between clients and servers allowing clients to access references of services from the servers. The clients need not know about the location of the servers or the services they offer. Similarly, the servers need not know the location of the clients that want to access the references of the services.

The reference of a service can be associated with properties that describe the service. The lookup service keeps a list of the registered references and their associated properties. These properties can be used by the Lookup service to select one or more services based on queries sent by the client.

To communicate with the Lookup service, the clients and servers need an access point. If the access point is not known, clients and servers use a bootstrapping protocol to find it. Typically a broadcast message is sent. The listening Lookup service responds with a message containing information about its access point.

Structure The following participants form the structure of the Lookup pattern:

A *service* provides some type of functionality.

A *client* uses a service.

A *lookup service* provides the capability for services to register themselves and for clients to find these services.

The following CRC[1]cards describe the responsibilities and collaborations of the participants.

Class Client	***Collaborator***
Responsibility • Uses a service	• Service • Lookup Service

Class Service	***Collaborator***
Responsibility • Provides application functionality	• Lookup Service

Class Lookup Service	***Collaborator***
Responsibility • Allows services to be registered • Allows client to find registered services • Associate properties with services	

1. Class-Responsibility-Collaborators (CRC) cards M. Stal, Activator Pattern, http://www.stal.de/articles.html, 2001 help to identify and specify objects or the components of an application in an informal way, especially in the early phases of software development. A CRC-card describes a component, an object, or a class of objects. The card consists of three fields that describe the name of the component, its responsibilities, and the names of other collaborating components.

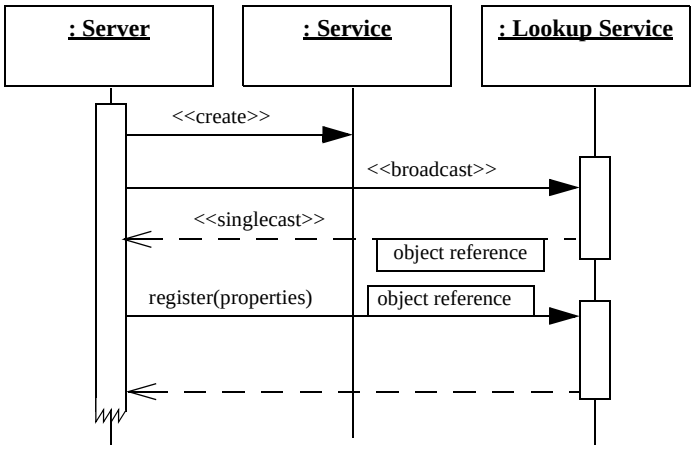
Dynamics There are two sets of interactions in the Lookup pattern. The first set comprises of registering a service with the Lookup service and includes the following interactions:

The server creates a new instance of a service.

It then searches for a Lookup service via a bootstrapping protocol, e.g., a broadcast protocol.

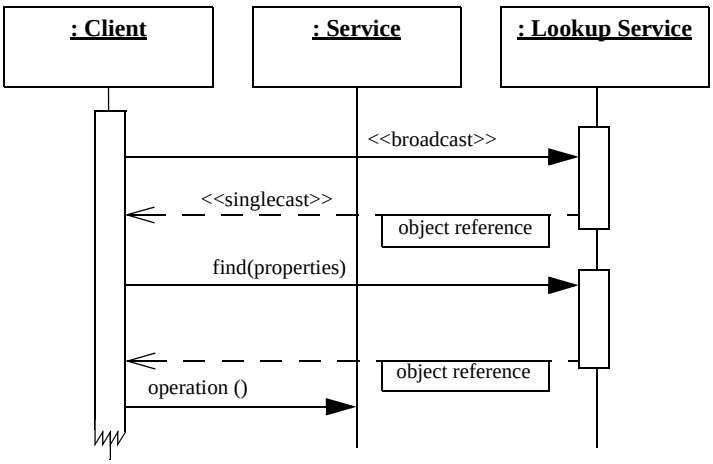
The Lookup service responds announcing its access point.

The server registers the object reference of the service using properties with the Lookup service.



The second set comprises of a client finding a service using a Lookup service and includes the following interactions:

- The client searches for a Lookup service via a bootstrapping protocol, e.g., a broadcast protocol.
- The Lookup service responds announcing its access point.
- The client queries the Lookup service for the object reference of the desired service using its properties.
- The Lookup service responds with the object reference of the desired service.
- The client uses the object reference to access the service.



Implementation There are four steps involved in implementing the Lookup pattern.

1 *Determine functionality of a Lookup service*. A lookup service should facilitate registration and lookup of services. It should provide an API that allows services to register and unregister themselves. Each registered service may provide meta information about itself that can be used by the lookup service to fetch the appropriate service upon client request. In the simplest case, the meta information may just contain the name of the service. Different policies can be defined for the Lookup service. For example, the Lookup service may support bindings with duplicate names or properties. The Lookup service should also provide an API that allows clients to retrieve a list of all available services as well as retrieve a reference to a particular service. The search criteria used by the clients can be a simple query-by-name or a more complex query mechanism as described in step 4 of the implementation section.

2 *Implement the Lookup service*. Internally, the Lookup service can be implemented in many different ways. For example, it may keep the registered services and their meta information in some kind of a tree data structure or a hashmap. The information itself may be transient or can be made persistent with an appropriate backend persistency mechanism.

> For example, Orbix 2000 [IONA] which is a CORBA 2.3 implementation uses the COS Persistent State Service to persist the name bindings in its Name Service. Other CORBA implementations such as TAO [TAO] persist the bindings using memory-mapped files.

3 *Provide the Lookup service access point*. The lookup service may provide a well-defined and well-known access point which can be published to the clients. The access point will typically include information such as the hostname and the port number where the Lookup service is running. This information can be published to the clients by several means such as writing it to a file that can be accessed by the client, or through well-defined environment variables.

> For example, a lot of CORBA implementations publish the access point of the Name Service using property or configuration files which can be accessed by clients.

If an access point is not provided by the Lookup service, it will be necessary to design a bootstrapping protocol which can allow clients to obtain the access point. Such a bootstrapping protocol is typically designed using a broadcast or a multicast protocol. The client sends an initial request for a reference to a lookup service using the bootstrapping protocol. The request contains information describing the type of request as well as the type of service, in this case lookup, that the client is interested in. On receiving the client's request, typically one or more lookup services send a reply back to the client passing along their access points. The client can then contact the lookup services directly to obtain references to other services.

> In CORBA, a client can get the access point of a Name Service using the `resolve_initial_references()` call on the ORB. Internally, the ORB uses a broadcast protocol to get the access point, an object reference, of the Name Service.

4 *Determine a query language*. The lookup service may optionally support a query language that allows clients to search for services using complex queries. For example, a query language could be based on using a property sheet that describes the type of service a client is interested in, in some cases even the actual data type of the service might be used. The properties pertaining to a particular service may be stored with the service itself or it may be stored externally, for example using a Property Service. The client when using the Lookup service to query a service may submit a list of properties

that should be satisfied by the requested service. The Lookup service can then compare the list of properties submitted by the client against the properties of the available services. If a match is found, the reference to the service is returned to the client.

> The CORBA Trading Service allows properties to be specified corresponding to a service that is registered with it. A client can build an arbitrarily complex query using a criteria that is matched against the properties of the registered services.

Example Resolved

Consider the example where a client wants to obtain an initial reference to a transaction manager in a distributed CORBA environment. Using the Lookup pattern, a Lookup service should be implemented. Most CORBA implementations provide such a Lookup service either in the form of a Name Service or a Trading Service, or both. These services are accessible via IIOP and provide well-defined CORBA interfaces.

In our example, the server which created a transaction manager should first obtain a reference to the Name Service and then use it to register the reference of the created transaction manager. The C++ code below shows how a server can obtain the reference to the Name Service and then register the transaction manager with it.

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Create a Transaction Manager
TransactionMgr_Impl *transactionMgrServant =
     new TransactionMgr_Impl;

// Get the CORBA object reference of it.
TransactionMgr_var transactionMgr =
     transactionMgrServant->_this();

// Get reference to the initial naming context
CORBA::Object_var obj = orb->
     resolve_initial_references("NameService");

// Narrow the reference
CosNaming::NamingContext_var ns =
     CosNaming::NamingContext::narrow(obj);

// Create the name with which the transaction
// manager will be bound
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Register transactionMgr object reference in the
// NS at the root context
ns->bind(name, transactionMgr);
```

Once the transaction manager has been registered with the Name Service, a client can obtain its object reference from the Name Service. The C++ code below shows how a client can obtain the reference of the transaction manager from the Name Service.

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get reference to the initial naming context
CORBA::Object_var obj = orb->
     resolve_initial_references("NameService");

// Narrow the reference
CosNaming::NamingContext_var ns =
     CosNaming::NamingContext::narrow(obj);

// Create the name with which the transactionMgr
// is bound in the NS
CosNaming::Name name;
```

```
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Resolve transactionMgr from the NS
CORBA::Object_var obj = ns->resolve(name);

// Narrow the object reference
TransactionMgr_var transactionMgr =
    TransactionMgr::_narrow(obj);
```

Once the initial reference to the transaction manager has been obtained by the client, it can then use it to invoke operations as well as to obtain references to other CORBA objects and services.

Variants Several instances of Lookup service can be used together to build a *federation* of Lookup services. The instances of Lookup services in a federation co-operate to provide clients with a wider spectrum of object references and services. A federated Lookup service can be configured to forward requests to other Lookup services when it can not fulfill the requests itself. This widens the scope of queries and allows a client to gain access to additional objects it was not able to reach before. The Lookup services in a federation can be in the same or different location domains.

Lookup service can be used to build fault-tolerant systems. Replication is a well-known concept in providing fault-tolerance and can be applied at two levels using Lookup service. First, the Lookup service itself can be replicated. Multiple instances of a Lookup service can serve to provide both load balancing as well as fault tolerance. The Proxy pattern [GHJV] can be used to hide the selection of a Lookup service from the client. For example, several ORB implementations provide smart proxies [OMG] on the client side which can be used to hide the selection of a particular Lookup service from among the replicated instances of all the Lookup Services.

Second, the services and objects that are registered with a Lookup service can also be replicated. A Lookup service can be extended to support multiple registrations of objects for the same list of properties, e.g. the same name, in the case of the CORBA Name Service. The Lookup service can be configured with various strategies [GHJV] to allow dispatch of the appropriate object upon request from a client. For example, a Lookup service could use a round-robin strategy to alternate between multiple instances of a transaction manager that are registered with it using the same list of properties. This type of replication is used by Inprise [INPRISE] to extend the scalability of their CORBA implementation called Visibroker.

Known Uses **CORBA**—The Common Object Services Interoperable Naming Service and Trading Service implement lookup services. Whereas the query language of the Name Service is quite simple, using just names, the query language of the Trading Service is powerful and can suit complex queries for components.

Java—The Java Naming and Directory Interface (JNDI) implements a Lookup service by providing directory and naming functionality to Java applications. Using JNDI, Java applications can store and retrieve named Java objects of any type. In addition, JNDI provides querying functionality by allowing clients to lookup Java objects using their attributes.

Jini—Jini supports ad-hoc networking by allowing services to join a network without requiring any pre-planning, installation, or human intervention and by allowing users to discover devices on the network. Jini services are registered with Jini's lookup service and these services are accessed by users using Jini's discovery protocol. To increase network reliability the Jini lookup service regularly broadcasts its availability to potential clients.

COM—The Windows Registry can be seen as some kind of lookup service. Clients know either the ProgId, the name and the version in some cases, or the GUID (Global Unique IDentifier) of the component. The registry allows then to retrieve the associated components.

DNS—The Domain Name Service is responsible for the coordination and mapping of domain names to and from IP numbers.

Telephone Directory Service—The Lookup pattern has a real world known use case in the form of telephone directory service. A person *X* may want to obtain the phone number of person *Y.* Assuming person *Y* has registered his/her phone number with a lookup service, in this case a telephone directory service, person *X* can then call this directory service and obtain the phone number of person *Y.* The telephone directory service will have a well-known phone number, for example 411, thus allowing person *X* to contact it.

Receptionist—Imagine someone is looking for another person, but just knows the house where that person is living. However, the other person does not live alone in that house. Now, if someone wants to talk to that person, he/she will ring the door bell. People in the house hearing the door bell ringing would know that somebody is at the door wanting to talk to them. However, it may not be clear to whom the person wants to talk to. So one of them, for example a receptionist, will answer the person ringing at the door. The receptionist forms an access point for the person. This allows him/her to ask for the person he/she is looking for and get a ’reference’ to that person.

Consequences There are several **benefits** of using the Lookup pattern:

Availability: Using the Lookup pattern, a client can find out on demand what distributed objects or services are available in its environment. Note that an object or service may no longer be available but its reference may not have been removed from the lookup service. Please see the Dangling references liability for further details.

Independence: The Lookup pattern allows a client to be able to obtain initial references of one or more distributed objects or services without relying on other unknown distributed objects or services. The well-known bootstrapping protocol allows the client to find the Lookup service and then use it to find other distributed objects and services.

Location Transparency: The Lookup pattern provides location transparency by shielding from the clients the location of the registered objects and services. Similarly, the pattern shields the location of the clients from the servers.

Configuration simplicity: Distributed systems based on a Lookup service need little or no manual configuration, no files need to get shared or transferred in order to distribute references to distributed objects. The usage of a bootstrapping protocol is a key feature for ad hoc networking scenarios, where the environment changes regularly and cannot be predetermined.

Property-based selection: References to distributed objects can be chosen based on properties. This allows more fine-grained selection of services including better matches between the client needs and the service offers.

There are some **liabilities** of using the Lookup pattern:

Single point of failure: One consequence of the Lookup pattern is the danger of constituting a single point of failure. If an instance of a Lookup service crashes, the distributed system can lose the registered references along with the associated properties. Once the Lookup service is restarted, the distributed objects would need to re-register with it unless the Lookup service has persistent state. This can be both tedious and error prone since it requires registered distributed objects to detect the Lookup service crashing and then restarting. In addition, a Lookup service can also act as a bottleneck and affect system

performance. A better solution, therefore, is to introduce replication of the Lookup service, as discussed in the variants section.

Dangling references: Another consequence of the Lookup pattern is the danger of having dangling references. The registered references in the Lookup service can become outdated as a result of their corresponding objects being terminated or being moved. In this case the Leasing pattern [LEASING], as applied in [JINI] can help by forcing the objects to prolong their 'lease' regularly if they do not want their entry removed automatically.

Unwanted replication: Problems can occur when similar objects with the same properties are registered but replication is not wanted. Depending upon the implementation of the Lookup service, multiple instances of the same object may get erroneously registered or one object may overwrite the registration of a previous object. Enforcing at least one of the properties be a unique identifier can avoid this problem.

See Also The Activator design pattern [ACTIVATOR] registers activated components with a lookup service in order to provide clients references to them. In many cases the references retrieved from a lookup service are actually references to factories, implementing the Factory design pattern [GHJV]. This decouples the location of components from their activation. The Lightweight Directory Access Protocol (LDAP) is a protocol for accessing Lookup services. It runs directly over TCP, and can be used to access a stand-alone LDAP directory service or to access a directory service that is back-ended by X.500.

Acknowledgements

We would like to thank our EuroPLoP 2000 shepherd, Bob Hanmer, for his feedback and valuable comments. We would also like to thank everyone at the writer's workshop at St. Martin, Austria during our Siemens retreat as well as the people of the writer's workshop at EuroPLoP 2000 for their comments and suggestions.

References

[ACTIVATOR] M. Stal, *Activator Pattern*, http://www.stal.de/articles.html, 2001

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

[GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[INPRISE] Inprise, http://www.inprise.com, 2001

[IONA] IONA, http://www.iona.com, 2001

[JINI] Jini™, http://www.sun.com/jini, 2001

[LEASING] P. Jain and M. Kircher, *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, August 13-16, 2000

[OMG] Object Management Group, http://www.omg.org, 2001

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects*, John Wiley and Sons, 2000

[TAO] The ACE ORB, http://www.cs.wustl.edu/~schmidt/TAO.html, 2001