

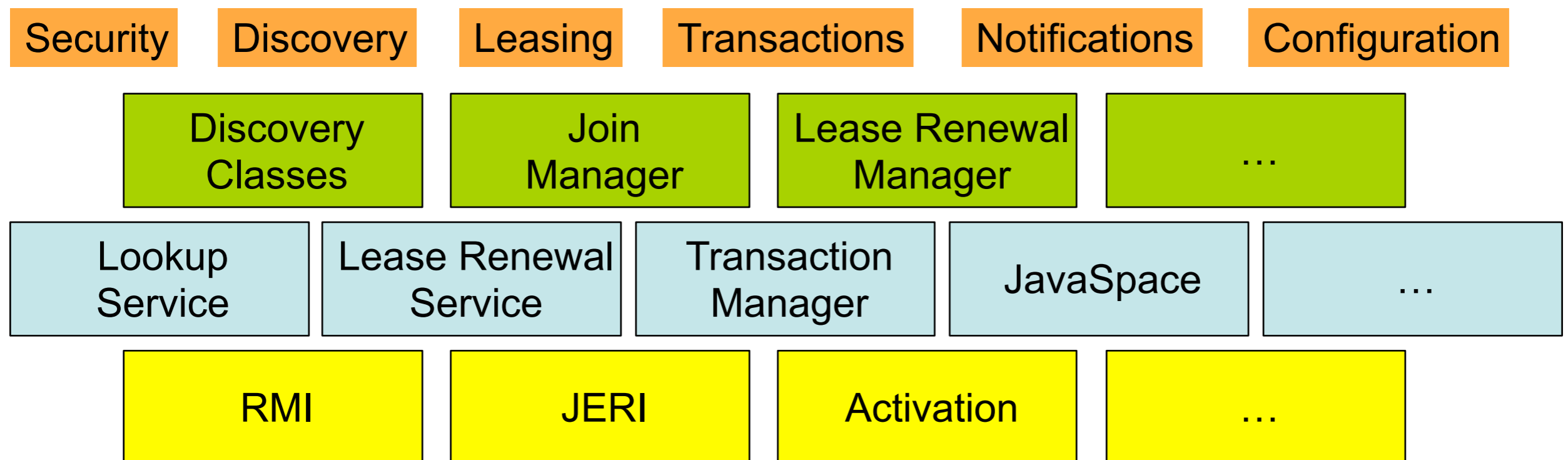
Jini

Outline

- Jini Architecture
- Discovery of services
- Leasing of resources
- JavaSpaces

Jini Architecture

- Jini is Sun's Java-based distributed platform focused on network dynamicity
- To help the realization of robust distributed systems Jini defines a set of specifications about discovering of services, leasing of resources, notification of events, distributed transactions, security etc.
- Sun also provides a Jini Reference implementation which implements part of these specification (as both services or helper classes for building yours)
 - The implementation is founded on top of basic Java technologies such as serialization, RMI, activation, distributed garbage collection, etc.
 - Many of this base services are extended by Jini
 - As an example Jini features JERI (Java Extensible Remote Invocation) which allows many customization over basic RMI



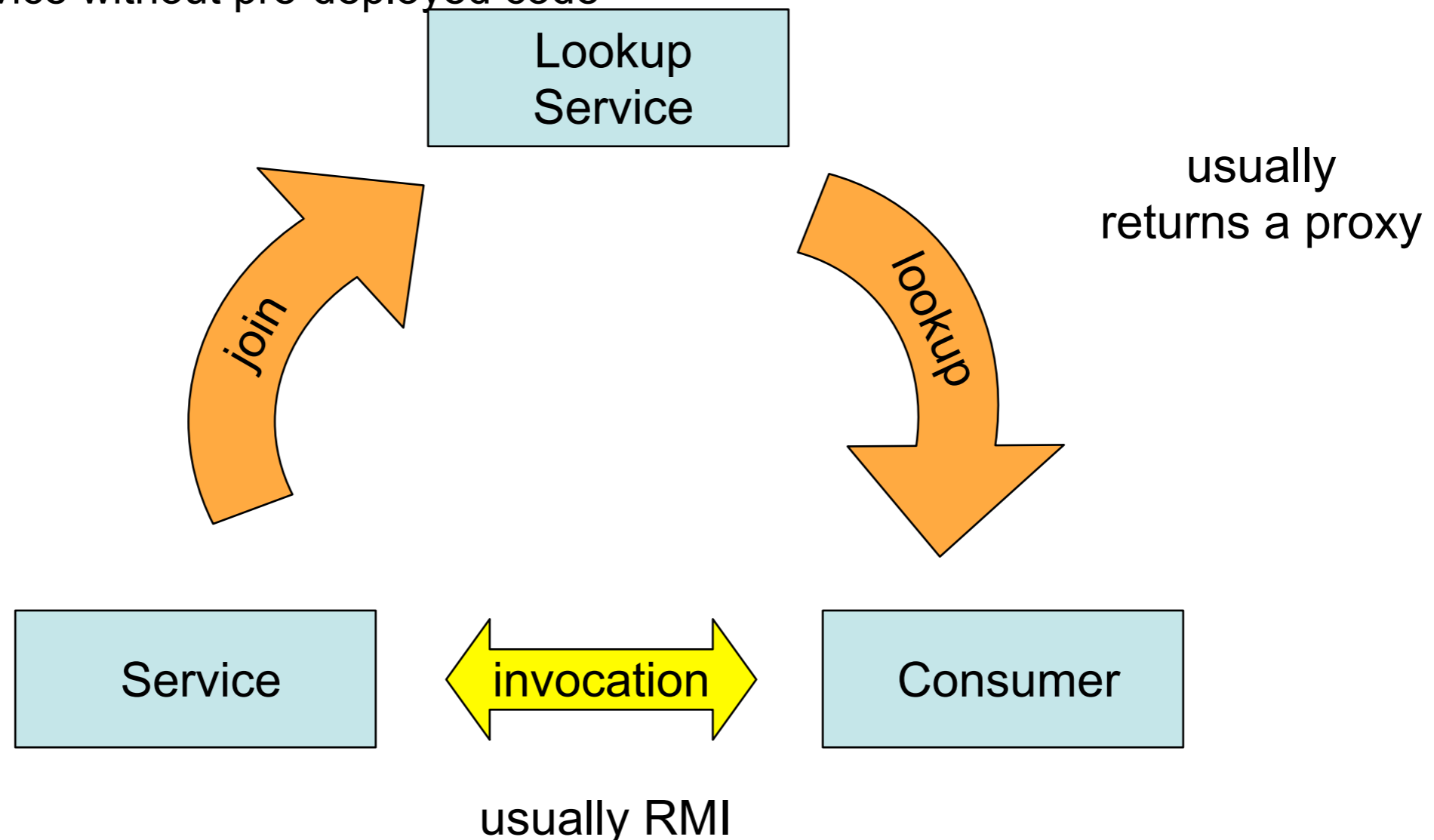
Jini Architecture

- Let's try to browse the services provided by the reference implementation
 - Launch all jini services with script `installverify/LaunchAll`
- We should be able to see
 - Lookup Service (Service Registrar): *reggie*
 - Persistence Service (JavaSpaces): *outrigger*
 - Transaction Service: *mahalo*

 - LookupDiscoveryService: *fiddler*
 - LeaseRenewalService: *norm*
 - Event Mailbox Service: *mercury*

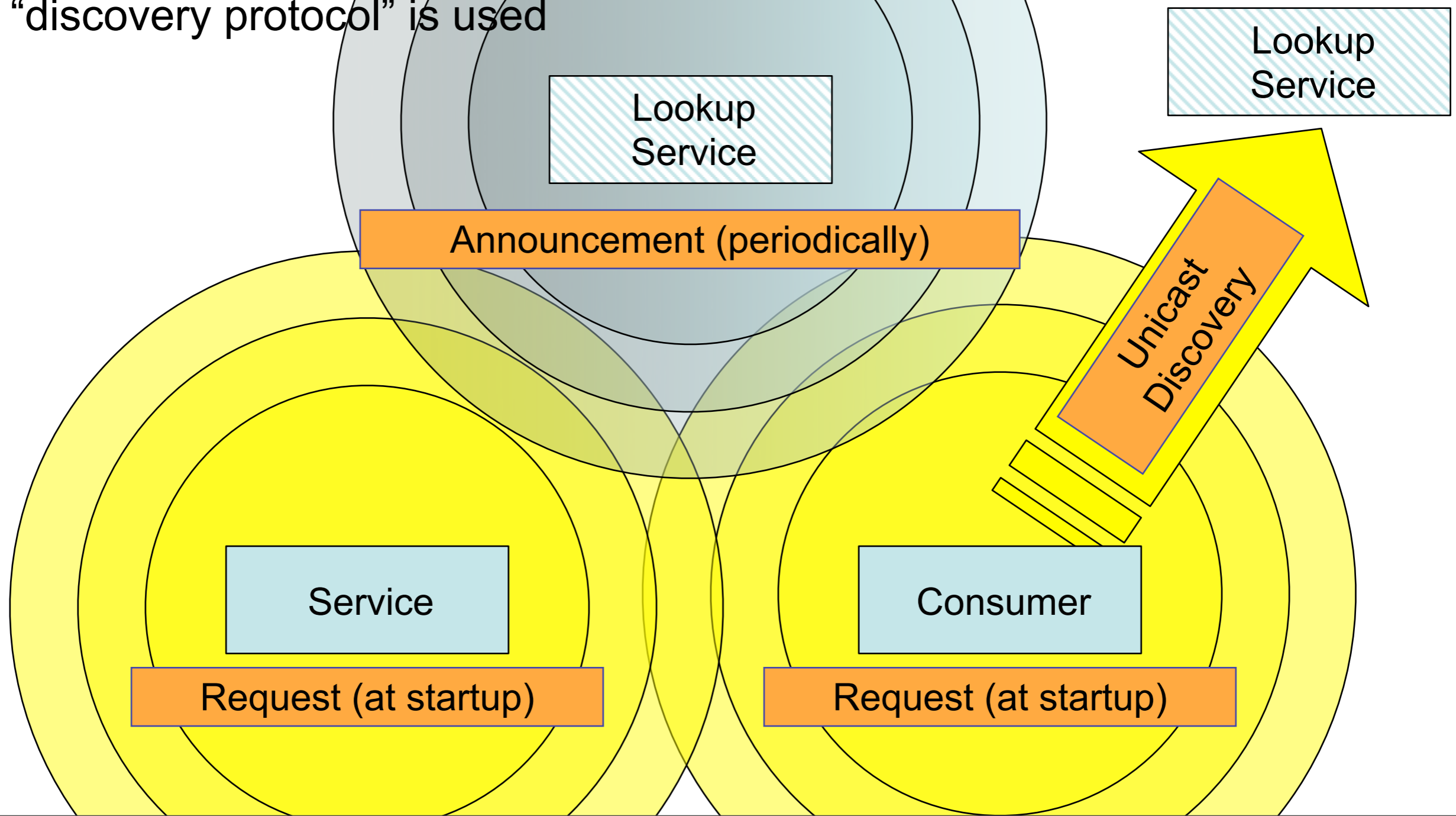
Typical Jini Interaction

- One of Jini's main goal is to seamlessly discover who is providing a service:
 - once this is achieved, the service is usually performed using RMI
- Discovery of services is aided by a Lookup Service (LUS)
 - You can think of it as an RMI registry for "resolving" interfaces instead of names
- Exploits mobile code: service proxies are dynamically downloaded, and enable the client to use a service without pre-deployed code



Discovery protocol

- The Lookup Service is essential for discovering other services
- To bootstrap the system (i.e. to discover the Lookup Services) a “discovery protocol” is used



Discovery protocol

- None of this protocol uses RMI
- The first message in each of the three protocols is a simple UDP packet
- In response to this UDP packet a TCP connection is created and a `net.jini.core.lookup.ServiceRegistrar` is transferred (a proxy of the Lookup Service) This transferral is the goal of the discovery
- Strings can be used to discover lookup service of a specific group

Discovery Classes

- Discovery Classes implements the discovery protocol
- LookupLocator
 - The simplest: it has a synchronous interface (unicast discovery)
 - Takes a String `url` (es “jini://bla.bla.bla”)

```
l = new LookupLocator("jini://localhost");
ServiceRegistrar lookupService = l.getRegistrar();

//Do something with the lookupService
...
```

Discovery Multicast

```
static public void main(String argv[]) {  
    new MulticastRegister();  
  
    // stay around long enough to receive replies  
    try {  
        Thread.currentThread().sleep(10000L);  
    } catch(java.lang.InterruptedException e) {  
        // do nothing  
    }  
}
```

Discovery Multicast

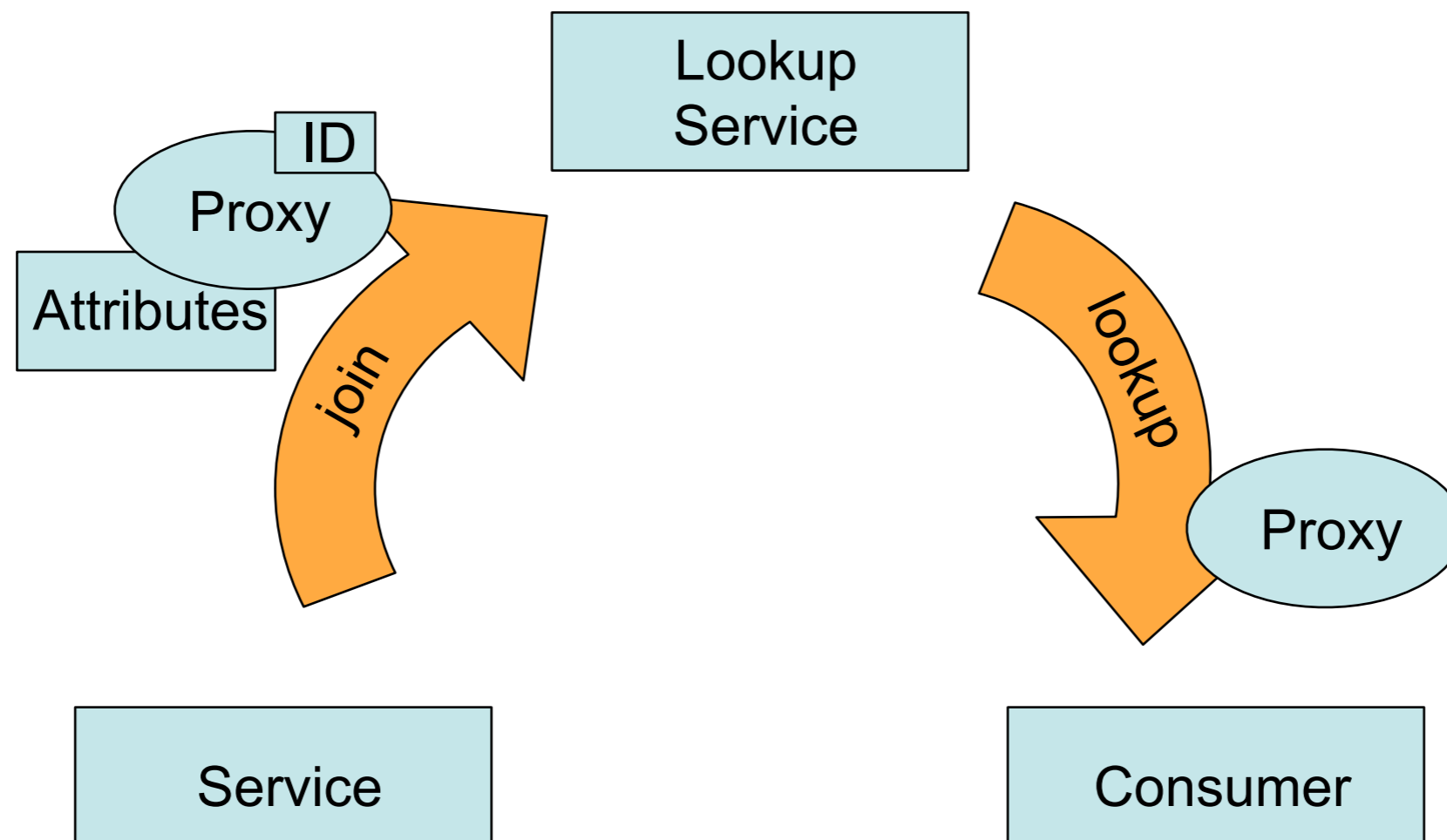
```
public MulticastRegister() {
    System.setSecurityManager(new java.rmi.RMISecurityManager());
    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch (Exception e) {
        System.err.println(e.toString());
        e.printStackTrace();
        System.exit(1);
    }
    discover.addDiscoveryListener(this);
}
```

Discovery Multicast

```
public void discovered(DiscoveryEvent evt) {  
  
    ServiceRegistrar[] registrars = evt.getRegistrars();  
  
    for (int n = 0; n < registrars.length; n++) {  
        ServiceRegistrar registrar = registrars[n];  
  
        // the code takes separate routes from here for client or service  
        try {  
            System.out.println("found a service locator at " +  
                               registrar.getLocator().getHost() +  
                               " at port " +  
                               registrar.getLocator().getPort());  
        } catch (RemoteException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Join/Lookup

- After obtaining a ServiceRegistrar (a stub to the lookup service) we can join or lookup a service



- `serviceRegistrar.register(...)` `serviceRegistrar.lookup(...)`
• Let's take a look at serviceRegistrar interface

Join Example

```
ServiceRegistrar lus = ...  
lus.register(new ServiceItem(id, service, null),  
             10000L);
```

ID Proxy Attributes

This is the lease period

This is the service id
null iff the service was
NEVER exported before

These are the attributes
describing the service

Important if you are registering a remote service you
should not register the object but the stub (use
UnicastRemoteObject.toStub)

Join Example Explained

- **ServiceRegistration register (ServiceItem item, long LeaseDuration)**
- **ServiceItem** contains
 - ***ServiceID serviceID***: null the first time a service is registered with the first lookup service. In following registrations the serviceID returned by the first ServiceRegistration should be used (even after a restart of the LUS or of the service itself)
 - ***Object service***: The serializable providing the service.
 - ***Entry[] attributeSet***: attributes describing the service: it can be null. We'll see Entries in a moment
- ***ServiceRegistration*** it contains a Lease (whose duration can be different from the required one)

Lookup Example

```
ServiceRegistrar lus = ...  
  
lus.lookup(new ServiceTemplate(id, new Class[]  
    {Interface.class}, null);
```

This is the
interface to
lookup

These are the attributes
describing the service

In case I want a specific
service instance

Lookup

- **Object lookup (ServiceTemplate tmp1)**
- **ServiceTemplate**
 - *ServiceID serviceID*: match if item.serviceID equals tmp1.serviceID (or if tmp1.serviceID is null);
 - *Object serviceTypes []*: match if item.service is an instance of every type in tmp1.serviceTypes
 - *Entry[] attributeSetTemplates*: match if item.attributeSets contains at least one matching entry for each entry template in tmp1.attributeSetTemplates
- A ServiceTemplate matches a Service if all of the three conditions hold

Leasing

- Service registration, like many Jini operations is subject to leasing
- Whenever an operation which is subject to a lease is invoked, the client requests a lease duration, the server (the lease grantor) could provide a lease with a shorter duration
- Operations under leasing (thus requiring a renewal) include:
 - Registering a service under a Lookup Service (LUS)
 - Writing tuples in a JavaSpace
 - Adding remote event listener
 - Transaction creation

Lease Renewal

- Lease interface

 - renew**, **cancel**, **getExpiration**

```
Lease l = ...  
l.renew(300000);  
l.cancel();
```

- A convenient way of dealing with leases is using a LeaseRenewalManager

 - renewFor**, **renewUntil**

```
Lease l = ...  
  
LeaseRenewalManager lmgr = new LeaseRenewalManager();  
lmgr.renewFor(l, Lease.FOREVER, 5000, null);
```

desiredDuration

renewalDuration

Lease Listener
called when
lease expires

JavaSpaces

- Generative communication
 - Write an item (OUT)
 - Read any one item given a template (READ)
 - Blocking or not blocking
 - Is not possible to read ALL matching items
 - Take an item (IN)
- JavaSpace also support Remote Events and Transactions

Entry

- Entries in Jini are used both to describe services and to write in JavaSpaces. It's a Java tuple
- An Entry must implement the tagging interface Entry (which extends Serializable)
- Fields of an entry are those of its members which are of **public** reference (i.e., Object) type
 - You cannot store primitive types in fields of an Entry, use wrapper objects instead
 - private/default fields are not considered
- An Entry must have an empty constructor (may have any number of other constructors or methods)
- Each field is serialized separately, so references between two fields of an entry will not be reconstituted to be shared references, but instead to separate copies of the original object

Matching of Entries

- Entries can be used also as *templates* for matching other entries
- An *entry* matches an *entry template* if
 - the class of the template is the same as, or a superclass of, the class of the entry
 - and every non-null field in the template equals the corresponding field of the entry
 - Every entry can be used to match more than one template. Note that in a service template, for `serviceTypes` and `attributeSetTemplates`, a null field is equivalent to an empty array; both represent a wildcard

JavaSpaces Api

- `Entry read(Entry tmpl, Transaction txn, long timeout)`
 - Read any matching entry from the space, blocking until one exists.
- `Entry readIfExists(Entry tmpl, Transaction txn, long timeout)`
 - Read any matching entry from the space, returning null if there is currently is none.
- `Entry take(Entry tmpl, Transaction txn, long timeout)`
 - Take a matching entry from the space, waiting until one exists
- `Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)`
 - Take a matching entry from the space, returning null if there is currently is none
- `Lease write(Entry entry, Transaction txn, long lease)`
 - Write a new entry into the space
- `EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback)`
 - When entries are written that match this template notify the given listener with a `RemoteEvent` that includes the handback object.