Designing Jini Distributed Services*

A Framework to support the development of reliable component networks

Marcelo B. D'Amorim
Universidade Federal de Pernambuco
Caixa Postal 7851, 50640-970
Recife-PE, Brazil
mbd@cin.ufpe.br

Carlos A. G. Ferraz
Universidade Federal de Pernambuco
Caixa Postal 7851, 50640-970
Recife-PE, Brazil
cagf@cin.ufpe.br

ABSTRACT

Resolving communication is not enough to address distribution because it is not the most difficult issue of developing distributed applications. The hard problems of distributed computing are not the problems of how to get things on and off the wire, but dealing with partial failure, lack of a central resource management, concurrency, and differences in memory access on local and remote resources [17]. Jini is a distribution platform that recognizes the differences between building stand-alone and really distributed systems. Its architecture provides a programming model, which enables developers to handle the hard aspects of distributed computing. However, Jini only provides a tool, and developers must apply such a tool to best address their needs to build distributed systems. This work presents the design and implementation of a framework aimed at building reliable Jini services on large-scale component networks.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software**]: Software Architectures—*Data abstraction, information hiding, patterns*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

Keywords

Component development, Separation of concerns, Jini.

1. INTRODUCTION

Designing distributed systems is harder than designing stand-alone and even client-server systems as distribution deals with aspects of concurrency, partial failure, availability, which do not hold or are simpler to deal with in local systems. Jini [7] is a platform for distributed application development which recognizes the differences between building monolithic and distributed applications. Some platforms such as CORBA make efforts to provide built-in support in order to isolate component of distributed concerns. Jini, in contrast, provides a programming model with which components must conform in order to deal explicitly with these aspects. By enabling programmers to deal with distributed aspects, Jini also introduces the burden of dealing with their complexity.

This paper presents the design of a toolkit to support the building of reliable distributed components. Automatic configuration [10, 13], for example, is specially supported by the toolkit in order to increase fault-tolerance in an environment where distributed services eventually crash while others become available. Actually, we assume in this work a component network like Ninja [14] or JTrader [3] is in place on a large-scale network in order to provide a service federation. By means of such federation different kinds of users could interact with different kinds of components using a semantics of *service trading*. Aspects such as administration interfaces, communication protocols, suitable UI (User Interfaces) to users interact with a service under different environments, and resource leasing have also to be carefully considered when developing distributed services.

This paper is organized in the following manner: Section 2 presents a brief tutorial on Jini technology, section 3 defines the design of a framework to support the implementation of Jini components. Beyond functional aspects, distributed components must address non-functional ones as well. Major non-functional issues such as service administration, communication protocols, user interfaces, resource leasing, and reconfiguration are so discussed in this section. Finally, section 4 concludes this work and introduce future advances.

2. JINI BACKGROUND

It is important to state that, in this section, we do not intend to extensively present the Jini's programming model and describe in detail main responsabilities that components must perform in order to get into a Jini network, but only a general background on its architecture and major concepts.

Sketched in Figure 1, the Jini architecture is organized into three categories: *infrastructure*, *programming model* and *services*. The infrastructure layer covers discovering commu-

---

[*] Paper presented on the First OOPSLA Workshop on Language Mechanisms for Programming Software Components held on Tampa Bay, Florida, 2001.

nities, joining services on these communities and searching for services, and it is responsible for providing minimal conditions for services to get into Jini networks. The programming model defines a set of API's that enables the construction of reliable services. While the first layer concerns with infrastructure issues such as service availability and location, this second layer covers application domain problems in a distributed context such as fault-tolerance (Leasing Service), asynchronous communication (Events) and distributed consistency (Transaction Service). The last layer holds services that make use of both the programming model as infrastructure. In fact, some of them have already been standardized, such as the JavaSpaces service [15]. The Jini architecture is very simple. Indeed, there is only one fundamental abstraction, that of a service. In this section, we describe this abstraction and a specific kind of service, the name service. A **service** is the Jini's central concept. A service may be hardware, software, or a combination of both. It implements an interface describing its behavior. This interface is required by the platform for every service, since it is the interaction point between the service and its clients. The implementation of a service, however, is only known to itself. More specifically, a service is described by three elements: an identifier, a proxy, and attributes. The identifier is assigned to the service when it starts, the proxy is a mobile entity which represents the service at the client, implements the interface describing the service, and isolates the communication protocol with the backend server from the client. Attributes provide additional description to the service (location, status, GUI and others). These three elements are represented by the `ServiceItem` class and are also called a **service offer**.

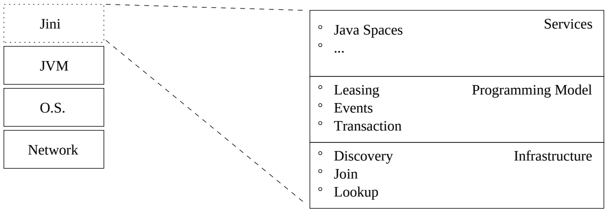

Figure 1: Jini architecure

Services associate themselves to form **communities** - also known as **groups**. A community is a logical entity represented by a `String` and reflects either the physical or the organizational structure of its services. For example, at the physical level, services in local network may form a community named "siteA"; at the organizational level, services in the marketing and management departments may be grouped in communities named "marketing" and "management".

Within a community, services interact with one another either as clients or servers. To support this interaction, they must get references to themselves, which is accomplished with support from a special service, the name service, known in Jini as the Lookup Service. This basic service describes the available services in a Jini community, providing operations for service search and registration. Before invoking these operations, however, a new service must get a reference to a Lookup Service that is actually attached to some community. This process is defined by the *Discovery* protocol [7].

In this protocol, a service does not need to know the location of the Lookup Service, which means that clients need no prior configuration to find the name service. An asynchronous protocol version, implemented with UDP multicast, searches for Lookup Service references within the local network radius. When it is found, a remote notification is sent back to the new service then the service is able to retrieve the Lookup Service proxy. The following code fragment illustrates the use of this protocol:

```
class Service {
 ...
 class Listener implements DiscoveryListener {
   public void discovered(DiscoveryEvent ev) {
     ServiceRegistrar[] lookupServices;
     lookupServices = ev.getRegistrars();
     ...
   }
   ...
 }
 ...
 LookupDiscovery disco =
   new LookupDiscovery(new String[] {"public"});
 disco.addDiscoveryListener(new Listener());
 ...
}
```

The protocol may be implemented by the class `LookupDiscovery`. When the `disco` object is created, an asynchronous search for a Lookup Service of the community named "public" begins in the local network. Then a `Listener` object is registered with `disco` so that when a Lookup Service is found, the `discovered` rook method is invoked to get a reference to this service. Note that the Jini Lookup Service implements the `ServiceRegistrar` interface.

Once a new service has a reference to the name server, it can register itself by invoking the Lookup Service's `register` method:

```
ServiceItem item =
  new ServiceItem(id, createProxy(), attributes);

ServiceRegistration sr =
  lookupService.register(item, LEASE_TIME);
```

The `item` argument represents the service. It provides an identifier to the service of `ServiceID` type, a proxy object and attributes (an array of `Entry` objects representing service properties). The second argument is a constant defining for how long the service offer registration is valid. The `sr` object is a record of the registration and, through the lease object that is enclosed on it, a service is able to renew its interest in keeping registered on the name service. In general, when registering in a Lookup Service, a service also follows a set of conventions, known as the *Join* protocol [7]. These conventions state that the service must renew its registration regularly and keep its attribute and identifier description consistent with every Lookup Service it registers itself with.

A service in a community may also look for another services through the use of a *Lookup* protocol [7]. The following code fragment shows the use of the —codeServiceRegistrar `lookup` method:

```
ServiceTemplate template =
  new ServiceTemplate(id, types, attributes);

Object serviceProxy =
  lookupService.lookup(template);
```

The `template` object specifies the service's identifier, the interfaces implemented by it, and its attributes. The result of the search is a service proxy matching all the data with the given template. Figure 2 illustrates interaction with the Lookup Service, also called Lus.

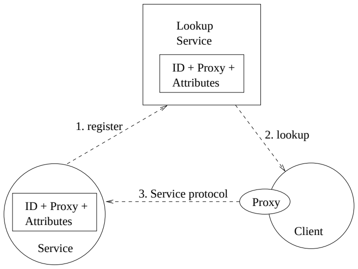

Figure 2: Lookup Service - Lus

3. TOOLKIT DESIGN

Designing software systems is hard since you must achieve a reasonable compromise between some conflicting aspects. Components must provide enough flexibility to cope with new requirements, but still be specific to the problem in hand. Designers must identify the right classes, hierarchies, and also configure the relationship among them by well balancing design issues. Design patterns [8] describe solutions to common problems in software projects. At a high abstraction level, patterns enable engineers to reuse solutions applied to problems already faced. The work of Beck et al. [1] provides a means by which a system architecture can be represented solely by patterns. It asserts that *patterns generate architectures* considering that they provide a high-level and sound model to derive and evolve architectures.

As a result of the JTrader system [3] development we observed that some patterns could be regularly applied when building Jini components. This suggested the development of a framework to support component development. There are some currently available, however, they do not consider the forces we are concerned with, specially that of separation of concerns. In general, these toolkits define a root class that encapsulates every aspect a service must deal with, and in turn user-defined services just extend this class and implement some rook methods. This approach works well when developing services on-the-small, but we experienced difficulties when components did not follow the behavior expected by the root class.

In order to be considered Jini citizens, **services** have to perform some key responsibilities [7]. For example, a service must store in persistent state its identification object - a `ServiceID` instance. This object, acquired at the first time the offer is registered, is to be used for registering the service offer in other Lus instances thereafter. Moreover, the service is in charge to renew its registration lease every time the lease is to expire. In fact, programmers have to deal with many issues: implement persistence, the *Discovery* and *Join* protocols, implement a protocol for smooth termination, define a specific service protocol that establishes how the proxy should contact the backend server or servers.

On the other hand, a **client** component is capable of trading for services in a Jini community by means of its Lookup Service. In fact, this capability enables unprecedented opportunity to dynamic software configuration, which is achieved by resolving dependencies among distributed components. As presented in [5], automatic configuration of distributed components is becoming an issue to provide resilience to services on large-scale component networks. As embedded systems and handheld devices, such as PDAs (Personal Digital Assistants) and cellular telephones, become popular, distributed adaptability deserves special attention of designers. For example, a Jini client component may stop running until it rediscovers a new instance of the failed service that provides the same interface. This situation may be an essential requirement as long as the client cannot perform its tasks without a reference to that service, and it is often referred to as *dependence management* [11].

On component networks, like Ninja [14] and JTrader [3], there is not the strict figure of clients and servers, but just components which are composed with other components. In fact, these components may perform both client and server roles. The framework is then supposed to support all these issues in order to aim the construction of distributed components, and its design is governed by the following forces:

- ⋆ *Separation of concerns* - Non-functional responsibilities that services have to perform are to be decoupled from their interfaces and implementation. This force aim at dealing with each service aspect independently.
- ⋆ *Ease of use* - The framework must simplify the implementation, configuration and deployment of Jini components.
- ⋆ *Minimal and complete interface* - The framework must provide a facade component by which programmers could interact with in order to implement their services. Since the service public interface concerns functional aspects only, the facade is the interaction point whereby non-functional aspects are configured.

This section is organized according to non-functional aspects a service must consider. Section 3.1 considers service administration and also presents the approach to configure dependencies between a user-defined service and its delegate components, which are charged to deal with different responsibilites of a service implementation. This approach makes use of a component called `Starter` that hides from the service implementation non-functional responsibilities. In sequence, section 3.2 considers communication issues, section 3.4 presents the framework leasing support to control resource misuse in the distributed environment. Finally, section 3.5 briefly describes the approach used by the framework to deal with reconfiguration.

3.1 Implementing Administration Interfaces

Jini services should implement administration interfaces in order to handle persistence, manage the discovery and join protocol, and also handle service termination by releasing all previously acquired resources. The Jini API defines interfaces which dictate how these tasks should be accomplished [7], like the `JoinAdmin` interface. In fact, Jini does not provide an exhaustive set of administration interfaces; there could be others, even user defined ones. Furthermore, Jini does not provide any implementation of these interfaces, beyond those of its own services. As a rule of thumb, services should implement these interfaces by themselves.

Implementing administration interfaces right on the service class may incur to some problems. This practice leads to an intricate implementation and non-reusable code since several distinct and in general not related aspects are programmed on a single software unit. The toolkit, on the other hand, approaches a solution that delegates to third-party components administrative responsibilities. As represented in figure 3, each administrative interface (`JoinAdmin`, `DestroyAdmin` and `StorageLocationAdmin`) is implemented by a delegate component, which has a default implementation provided by the toolkit. These components have dependencies among each other. For example, the `JoinDelegate` should notify the `StorageDelegate` when the join state has changed, like when the set of groups to which a service joins is being modified. Also, the `DestroyDelegate` should notify any resource consumer delegate that the service is about to be destroyed. In this case, `StorageDelegate` would have to checkpoint the service state and also close non-memory resources acquired, like an opened file, for example. As represented by the `DestroyListener` and `CheckpointListener`, the toolkit uses a slight different variation of the *Observer* design pattern [8] to resolve these dependencies. In its most common version, the subject of observation (observed) calls back the subject entity (observer) when a state change has taken place. In contrast, toolkit's delegators communicate the observed state along with the notification using an approach very similar to the Java event model. Actually, this pattern is applied twice. When a service terminates, no object is to be transmitted to observers within the notification. However when some modification is made on the service state, a `PersistentObject` should be communicated.

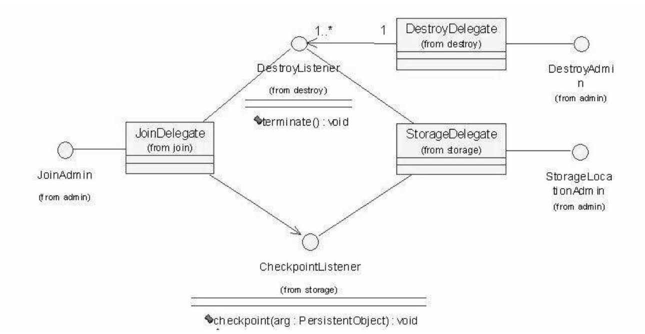

Figure 3: Dependencies among delegate components

This approach uses N (where N is the number of observers registered to receive notifications) method calls to notify observers and also communicate some state change; in contrast with the original observer pattern, which uses 2*N messages due to "callback". Despite this advantage, the decision for this approach was justified for simplicity. Applying this pattern with distributed observers may reduce the number of messages to one because of multicast communication. However, we did not envision a scenario where the administration objects are remote to the service object they administer.

Introducing Persistence

The Jini `StorageLocationAdmin` interface defines operations to set and get the current storage location, suggesting that Jini's support to persistence uses the file system, however, this interface does not define operations to store and retrieve the service state from persistent storage. These operations are required to update the persistent state of the service - *checkpoint* [12, 6] - and also to restore such state prior to the service becomes active - *backward recovery*. The `StorageDelegate` toolkit's class indeed defines these operations. Therefore, in contrast with the other delegator components, the relation between the Jini interface and the `StorageDelegate` is not equivalent because the Jini API does not define a complete set of operations on a single interface to implement persistence. The persistent representation of a service supported by the toolkit is not supposed to be hierarchical like an ordinary object, but flat. As depicted on the figure 4, a service is represented persistently by an object of `CompoundPersistentObject` type that maps `Class` instances to `PersistentObject` instances and, in principle, any object is able to persist its state through the storage delegate. For example, a `JoinState` object is built by the `JoinDelegate` component prior to "checkpoint" its state to the `StorageDelegate` of a given service. The "checkpoint" event is triggered when the `JoinDelegate` component verifies that its state has been modified.

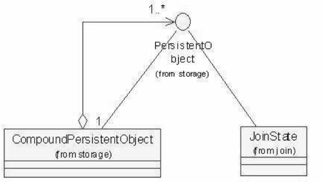

Figure 4: Composite

As the delegate approach attempts to separate concerns in independent components, such decoupled (flat) representation of the service state suits well as it allows any object, even those not concerned on the first design, to be included as a participant on the service state without affecting other dependant components, and also allow the state of a service to be distributed on classes related to its semantic. As an example, the `JoinState` is a fundamental toolkit class since it represents the persistent elements necessary to enable discovery and join protocols. The JoinState is the part of the service state that comprises, for example, the identification of the service, a list of properties, and a list of groups where the service should be registered.

The Java RMI activatable framework provides support to long-lived persistent objects. In essence, it means that a service is able to restore its state after a failure and continue servicing without affecting client remote references. Using `UnicastRemoteObject` turns a remote reference invalid whenever the server fails. In addition, by using the activatable alternative instead, the storage delegate has the opportunity to retrieve the service's state from the persistence mechanism after the underlying framework notifies a failure recovery event.

Configuring Dependencies among Delegate Objects

Isolating the implementation of administration interfaces in specific classes gives rise to other problems: how and where these administration delegate objects are created and how the service should be aware of them. In order to answer the first two questions we recur to the *Abstract Factory* pattern [8]. The remaining question we defer to the following section. The abstract factory pattern hides from the programmer how concrete classes are instantiated and what relationships do exist among created objects. In addition, the factory provides methods to retrieve the objects properly configured. The `DefaultAdminFactoryImpl` class, illustrated in figure 5, represents the default toolkit administration factory implementation, but users are completely free to develop new ones which better suit to their purpose. In practice, any `AdminFactory` implementation is valid.

Recall that the toolkit delegate components have dependencies among each other. Hence, a factory can express these dependencies internally so that clients only interact with objects returned from factory methods; and these objects are abstract type implementations, rather than concrete ones. System administrators should rely on Jini administrative interfaces, not concrete implementations, and therefore the `AdminFactory` is in charge of creating these Jini types. Except for the case of `IStorageDelegate`, all objects created by the factory are direct implementations of the administrative interfaces. As mentioned before, this happens because the Jini `StorageLocationAdmin` interface does not provide a complete set of operations required to implement persistence. However, these additional capabilities implemented by the `IStorageDelegate` (checkpointing and recovering) are hidden from the admistrator.

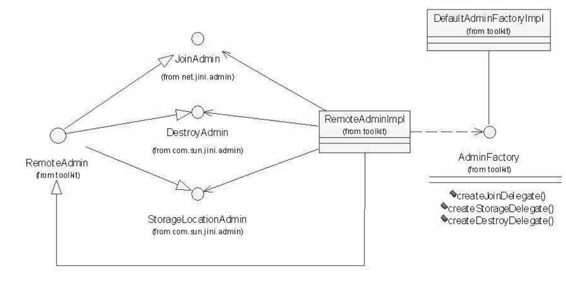

Figure 5: Abstract Factory

Considering that services regularly implement the basic Jini administrative interfaces, the `RemoteAdmin` is defined as a means to unify these functionalities in just one remote component, which is the server-side counterpart for administration. This is desired to avoid the construction of it n different remote types for administrative purposes. In fact, beyond the administrative interfaces, this type also extends `java.rmi.Remote` interface turning its implementations accessible in remote locations. The `RemoteAdmin` concrete class - `RemoteAdminImpl` - implements these interfaces by delegating its responsibilities to the objects the `AdminFactory` creates. This behavior is represented in figure 5 by the `RemoteAdminImpl` outgoing arrows. This administrative backend server is actually the object to be returned when calling the `getAdmin` method on some `Administrable` type.

In addition, as an application typically needs only one instance of a concrete factory per family of created objects, `AdminFactory` concrete classes may also apply the *Singleton* design pattern [8]. In what follows, we present how a service is configured to use the backend administration component.

Resolving Service's Dependencies

Now we define a service example to present main dependencies between some service and the toolkit. Basically, this example defines three components, which are represented in figure 6: the `IPatternExample` interface, the `PatternExampleProxy` class, and the `PatternExample` class, which is actually the service implementation class. A service has two dependencies on the toolkit:

- *IService*- Implemented by the service class, this interface defines the `getServiceProxy` method used to retrieve the service proxy object.

- *Starter*- This central component is responsible to export the remote object to the RMI system[1], starts the administration process, and also loads the service configuration file. Actually it behaves like a facade object [8, 4] between the service and the toolkit.

The `Starter` component creates a `RemoteAdmin` object and defines a method to return the proxy responsible to remote control this administration backend object - notice in figure 6 that, like the service implementation, the `Starter` actually implements the `Administrable` interface. When the `getAdmin` method is called on the service, it calls the `Starter getAdmin` method in turn. Moreover, in order to enable user-defined factories to be easily configured in customized services, the `Starter` component loads a factory object by reflection[2] and passes it as parameter to the `RemoteAdminImpl` constructor. This special parameter is retrieved from a configuration file named "<service name>.properties" which is stored on a `Properties` object accessible through the `getParameters` method of the `Starter` class. However, if a user-defined factory is not set, the default one aforementioned is used instead.

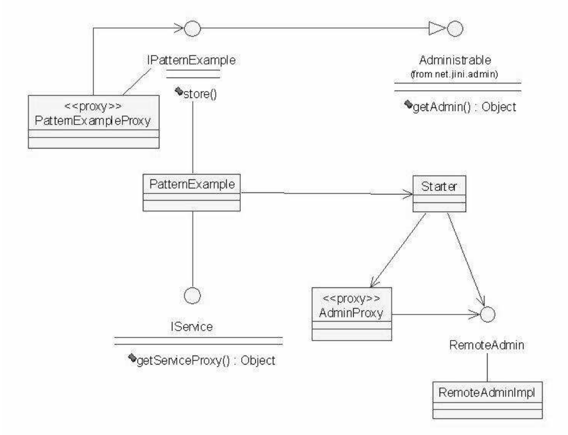

Figure 6: Toolkit components and dependencies.

3.2 Dealing with Communication

The Jini architecture defines an object located on the same address space as the client which behaves as a service counterpart. As presented on the section 2, this is the service proxy. Therefore, any distribution platform can be selected to handle communication in principle; that is, when the service is not entirely locally implemented. Despite this capability, the toolkit starter component currently supports only RMI but it does not enforce users to use it.

---

[1] In principle, Jini can be implemented over any distributed platform.

[2] Reflection is a Java API that provides access to a class representation. For example, its constructors and methods.

Minimizing implementation efforts due to communication is a very important toolkit concern in order to achieve productivity. In RMI, concrete `Remote` object implementations extends `UnicastRemoteObject` or `Activatable`. Extending the former class enables objects to register on the RMI system and be able to receive remote calls. However, if the client fails to communicate with the object due to a machine failure, for example, that client must get a new reference to the remote object. On the other hand, implementing activatable objects provides a means for persistent remote references thus, if a remote object fails, binding reconfiguration occurs transparently when the service recovers. After the remote component turns back running, remote calls through the activatable reference are likely to succeed since the remote reference is persists to failures.

The toolkit assumes that the Jini `ServiceStarter` utility class (this is not the toolkit `Starter` class) is used to ease bookkeeping of activatable objects. This component is in charge to create activatable groups, an activatable identification, and then call the activatable service constructor by reflection. Although it does not extend `Activatable`, the service behaves as if it had really extended it since; as mentioned, the constructor is supposed to be called by reflection thus relaxing the strong typed characteristic of Java. In addition, by using reflection we eliminate a dependency between the activation framework and the service. The following code illustrates how communication an the service initialization proceeds:

```
public class PatternExample
 implements IPatternExample,IService {

    Starter starter;

    public PatternExample(ActivationID id,
                          MarshalledObject state)
      throws IOException, ClassNotFoundException {

        starter = new Starter(this, id, state);
    }

    public Object getServiceProxy() {

       return new PatternExampleProxy((Remote) this);
    }
    ...
}
```

The service object passed as the first parameter to the `Starter` constructor must implement the `IService` interface. Within this constructor, the service remote object should be registered on the RMI system. In addition, the service proxy should be retrieved in order to start the administration process, and the service state, properly represented by the `state`[3] object have to be restored. Notice that the following code fragment does not present the factory object reflective load. However, when a customized factory cannot be loaded, the default factory implementation is used instead.

```
....
Activatable.exportObject((Remote) service, id, 0);
proxy = ((IService) service).getServiceProxy();
if (factory == null) {
    factory = new DefaultAdminFactoryImpl(proxy);
}
adminBackend = new RemoteAdminImpl(factory);
....
```

When using the Jini `ServiceStarter` class to start an activatable service, it is required to supply the groups that the service should join. Therefore, after the reflective construction of the service object, such utility class retrieves the `Administrable` object by calling the service `getAdmin` method and test, using the `instanceof` operator, if it implements the `JoinAdmin` interface. If so, the service is automatically registered on Lookup Service that participates on informed groups. As a consequence, this event causes the service persistent state to be updated (due to the dependence among delegates) and it becomes available on the network (due to the `JoinDelegate`).

3.3 Supporting User Interfaces

Very often applications merge user interface (UI) and functional code, turning reuse difficult as well as the maintenance of graphical components and business rules. Jini services are able to provide their own user interfaces and they do that by separating user interface and functional concerns in different modules. The UI behaves like a human being *adapter* [8] as it provides access to the service by a distinguished interface. This can be a great advantage concerning interoperability. The UI knows the service interface and how to interact with it by means of a proxy. This way we enable interoperability with the final user, not among services. To achieve that user interfaces need to be built on the client address space. Code transmission is so required not only for the service proxy but also for the UI.

The user interface can be a Swing component, AWT (Abstract Windowing Toolkit), textual, speech-controlled, or whatever the service designer wishes. The word UI is used intentionally to distinguish from GUI as it does not stand only for graphical components [16]. A service may have as many UI components as `UIDescriptor` objects it contains. As a subclass of `Entry`, the objects such descriptor represents are regular service properties, which are bound to a `ServiceItem` object during service registration (see section 2). In addition to strictly describing user interfaces, descriptors provide access to a factory object enabled to create a concrete user interface object. UI enabled services are in charge of creating the descriptor object and attaching it to the service offer as a property. If the client environment is capable of supporting the graphical toolkit required by the UI descriptor, say some version of the Java Swing, it requests the factory to create the user interface by passing the service proxy as parameter to the factory method.

In fact, the user interface aspect is well isolated on the descriptor object. Third-party components could even be acquired and used as customized user interfaces as long as these components relies on the same interface the service implements.

In order to support user interfaces and isolate the service class from administrative interfaces, the `Starter` component defines a public `addEntry` method.

```
public void addEntry(Entry entry) {
    JoinAdmin admin = (JoinAdmin) adminBackend;
    Entry[] entries = new Entry[]{entry};
    admin.addLookupAttributes(entries);
}
```

An instance of `UIDescriptor` can be passed as parameter to this method, which uses the `JoinAdmin` object to add properties to the service offer. Therefore, services can build a UI descriptor and then configure it as an eligible user interface by calling this method, defined in the `Starter` component.

---

[3] A `MarshalledObject` enables object compression.

3.4 Leasing Resources

Lease [9] is a powerful mechanism provided by the Jini programming model, which establishes that resource allocation interest should be renewed in order to control misuse. Events and Lookup are elements of Jini architecture that use leases to manage respectively event and service registration. For example, a service, which registers to receive some kind of event, must renew the lease returned within the event registration while it is interested in receiving those events. Two components participate on the lease "protocol": the lease grantor and the holder. The former is in charge to manage the resource allocation and return a lease that must be renewed by the later in order to guarantee access to the resource. When using events, the resource the lease grantor stores is actually the lease holder's interest in receiving events. Remarkably, the holder's role is supposed to be performed by services, however this is not a requirement [2].

Lease grantors must honor some Jini specification rules like: "do not renew a lease for more than the requested time", "throw `UnknowLeaseException` if the holder tries to renew or cancel the lease after expiration", etc. Therefore, developing lease grantors is harder than developing holders and may be well supported by the toolkit.

Providing methods to cancel and renew leases, the `Landlord` is the Jini remote interface grantors should implement. A landlord instance behaves as a server to every lease instance the grantor generates. The toolkit provides an abstract implementation of this interface through the `toolkit.lease .AbstractLandlord` class which is in charge of implementing the operations that do not depend on the mechanism used to store the leased resource, leaving to concrete subclasses the implementation of the storage mechanism. In fact, the toolkit provides a concrete and transient (non-persistent) `AbstractLandlord` implementation - `toolkit.lease.HashtableLandlord` - which uses a hashtable to store leased resources, instances of the `LeasedResource` Jini's interface. This type defines methods to get the leased resource identification (also called "cookie"), set and get the lease expiration time. Figure 7 presents the abstract landlord class and its main dependencies. The abstract landlord delegates the lease object creation to a factory class, represented by the Jini class `LandlordLeaseFactory`. The factory is able to create `LandlordLease` (a subclass of `Lease`) instances that contain the identification of the leased resource and a reference to the remote landlord. Therefore, this object serves to lease holders contact grantors, say landlords.

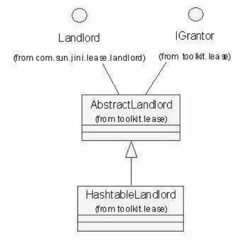

Figure 7: The abstract landlord

The Jini API defines what operation landlords should perform to manage lease renewal, however, it does not define operations through some interface to **create** leases. The `toolkit.lease.IGrantor` interface, presented below, does exactly this. Notice that `AbstractLandlord` implements both `Landlord` and `IGrantor` interfaces, so it is able not only to manage lease renewal but also granting leases.

```
public interface IGrantor {

   public Lease leaseFreeSlot(Object value,
                              long duration)
     throws LeaseDeniedException, RemoteException;
}
```

The `leaseFreeSlot` method requests space to allocate the resource, represented by the `Object` parameter. As a result, a lease with an expiration time not higher than `duration` miliseconds is returned to the holder, which is in charge to properly renew the lease. In order to conserve service encapsulation, lease holders should not access methods defined on the `IGrantor` interface directly. Services very often provide specific operations which indeed lease resources, thus hiding the `IGrantor` interface to the client. For example, the `ServiceRegistrar` Jini interface defines a `register` method that returns a lease within a service registration object. In such case, the Lookup Service could have used the toolkit abstract landlord to implement the `register` operation once the `IGrantor` interface provides a general functionality.

3.4.1 Leasing Service's Resources

Following to the example presented in the previous subsection, now we introduce leasing to the `PatternExample` service. The principle of isolating concerns still applies, so that functional aspects are implemented in the service and non-functional aspects in delegated objects; notice the abstract landlord is the object in charge to manage leases. In contrast to administration delegates, which are completely isolated from the service functionality, the service class depends on a lease delegate object since the request to create a lease is triggered by its own in response to a client request. As described earlier, the Lookup Service `register` method is an example of such behavior. After the lease object is created, communication conveys directly through the landlord, thus alleviating the service of lease bookkeeping. Therefore, the interaction between the service and landlord is regularly represented by the `IGrantor` interface, as presented in the figure 8. The `Starter` component stores a reference to an `AbstractLandlord` object, provides an operation to get a reference to the `IGrantor` interface, and an operation to set the abstract landlord to be used. If the service does not set one, the default implementation - `HashtableLandlord` - is used instead.

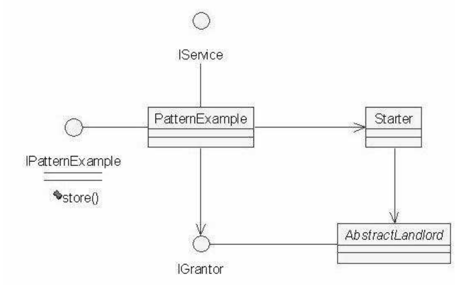

Figure 8: The service and landlord interaction

Suppose the `IPatternExample` defines an operation, named `storeMessage`, that allows clients to store a message remotely. This operation returns a registration object which contains an enclosed lease object. The following code implements this example:

```
public class PatternExample
 implements IPatternExample, IService {

  Starter starter;
  ...
  public MessageRegistration store(String message,
                                    int duration)
      throws RemoteException, LeaseDeniedException {

   AbstractLandlord absLandlord =
     starter.getAbstractLandlord();
   IGrantor grantor = (IGrantor) absLandlord;
   Lease lease = grantor.leaseFreeSlot(message, duration);
   return new MessageRegistration(lease);
  }
}
```

Neither the client nor the service interacts directly with the Landlord. The service calls the `IGrantor` to create lease objects, the client calls the `Lease` when it is to be renewed, and the lease calls the `Landlord` through the enclosed reference stored within the lease object.

3.5 Fault-detection and Reconfiguration

The Reverse Lease Subscriber (RLS) design pattern [2] proposes a means to detect failures and reconfigure service dependencies on large-scale component networks that provides a trading semantics to enable service discovery. In fact, both pattern and toolkit are integral part of the JTrader efforts to build a global service federation on the Internet. This toolkit uses the pattern to increase reliability of composite services. An extensive discussion about reconfiguration on component networks and about the RLS pattern is given on [2].

4. CONCLUDING REMARKS AND FUTURE WORKS

This paper presented the design of a framework to support the development of Jini components that perform both client and server tasks. Instead of concentrating responsibilities on the service class, specialized delegate components are instructed to deal with non-functional aspects thus isolating from the functional code these concerns. As long as aspects as communication protocols and fault-tolerance deserve very special care in distributed systems, we argue for the relevance of this approach. We observed that services communicate with delegate components in a disciplined manner. A discipline imposed by the `Starter` component that behaves like a facade object.

The current framework version does not support transaction management and event producers. In addition, it lacks full support to fault-tolerance. The framework helps on automatically searching currently available components that implement a similar interface as that of a failed service (see section 3.5). It resolves the problem under a client point of view, but it still lacks support to enable fault-tolerance at the server-side. Issues like election protocols, active replication, and implementation of shared state [12, 6] shoud be considered in future releases.

5. REFERENCES

[1] K. Beck and R. Johnson. Patterns Generate Architectures. In M. Tokoro and R. Pareschi, editors, *Proc. European Conf. on Object Oriented Programming (ECOOP)*, volume 821, pages 139–149, Bologna, Italy, 1994. Springer-Verlag, Berlin.

[2] Marcelo B. d'Amorim. Reverse Lease Subscriber: a Design Pattern for Failure Detection and Reconfiguration of Distributed Systems. In *proceedings of the First Latin American Conference on Pattern Languages of Programming - SugarLoafPLoP'2001. Rio de Janeiro-BRAZIL.*, 3th-5th October 2001.

[3] Marcelo B. d'Amorim. Service Trading on the Internet, the JTrader approach, May 2001. Univ. Federal de Pernambuco. M.Sc. dissertation.

[4] Douglas Schmidt et al. *Pattern-Oriented Software Architecture: Patterns for Concurrency and Networked Objects.* John Wiley & Sons, September 2000.

[5] Fabio Kon et al. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems - COOTS'2001*, Texas, February 2001.

[6] George Coulouris et al. *Distributed Systems: Concepts and Design.* Addison-Wesley, 2th edition, May 1994.

[7] Ken Arnold et al. *The Jini Specification.* Addison-Wesley, December 1999.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object Oriented Software.* Addison-Wesley, Jan. 1995.

[9] Prashant Jain and Michael Kircher. Leasing. *Pattern Language of Programming - PLOP'2000. Allerton Park, Monticello, Illinois, USA*, 13th–16th Aug. 1996.

[10] Prashant Jain and Douglas C. Schmidt. Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services. In *3rd USENIX Annual Pattern Languages of Programming Conference, Allerton Park, Illinois*, pages 209–219, 1997.

[11] Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26–36, January-March 2000.

[12] Evan Marcus and Hal Stern. *Blueprints for High Availability: Designing Resilient Distributed Systems.* John Wiley & Sons, January 2000.

[13] Francisco Assis Rosa and Antonio Rito Silva. Component Configurer. In *Proceedings of the 2nd European Conference on Pattern Languages of Programming - EuroPLoP '97*, pages 209–219, 1997.

[14] Steven D. Gribble et. al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of IEEE Computer Networks on Pervasive Computing (to appear)*, 2000.

[15] Sun Microsystems. *JavaSpaces Service Specification*, 1.1 edition, October 2000.

[16] Sun Microsystems and Artima.com. *The Jini User Interface Specification*, April 2000. Available at http://artima.com/jini.

[17] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *Note on Distributed Computing.* Sun Microsystems Laboratories, November 1994.