User Interfaces for Jini Services

Vrijendra Gokhale
vrigo@it.iitb.ernet.in

Bernard Menezes
bernard@it.iitb.ernet.in

Abstract

Attaching a user interface (UI) to a jini service can enable a context aware computing environment. If all jini services can create UI on client devices of varying UI technologies then clients can simply join a local jini federation, ask services to create UI and interact with them. This can be achieved if all jini services implement a common well known java interface that facilitates UI construction. UI construction also must adapt to varying client device technologies. This paper presents a pattern based approach towards solving this problem.

1 Introduction

Today microprocessors are becoming increasingly pervasive in our everyday lives. Appliances around us are getting more and more intelligent. With the advances in wireless technologies, the future networks would be seamless, dynamic and all pervading. Ideally they would be jini communities where services dynamically join and leave. Clients would join such federations to avail the services in their proximity.

Context awareness is the ability of the client to control, access and compute-using the services around him[1]. In order to enable this kind of serendipitous interaction to any services found in a local jini federation clients need a standard way of interacting with the services. This task is difficult as clients cannot predict what services they might discover in such dynamic situations. The solution is to have a generic interface implemented by all jini services. This interface must also allow the client to access the service specific functionality! These requirements are conflicting and how they can be met is the import of this paper.

2 User Interfaces for Jini services

The solution is to attach a user iterface (UI) to every jini service. All jini services may implement a common, well-known interface called vrigo.core.JUnknown. JUnknown has a method which initiates the UI construction. The UI construcion is service specific and would represent the service specific semantics. The key idea is to let a service stub directly talk to a user by creating a user interface.

Thus a vrigo.radio.Radio stub, when discovered by a client PDA as a JUnknown would paint a GUI consisting of sliders and spin controls that let the user manipulate and control the volume and frequency for the radio service. Similarly a toaster stub would display a choice list for selecting from a set of toasting options. Here the client application does not programatically talk to the radio or the toaster interface. Instead it calls a method of the vrigo.core.JUnknown which initiates the UI construction. The user can then directly interact with the service using the UI.

3 Using patterns

Client devices may have different UI technologies. Thus Jini service stubs should have the capability of discovering client UI capabilities and should interact accordingly with the user. Thus if the client device is a web browser the UI would be HTML content. If the client has a simple text based display then a text based UI should be created. Note that a GUI is just another form of a User Interface(UI).

With varying display technologies of the clients, the UI construcion logic would also differ! A text based UI would respond differently than a voice response based UI to user interactions. This construcion logic needs to be dynamically loaded into the client JVM based upon the client UI capability.

These two factors motivate the use of the Factory and the Builder patterns[2]. To encapsulate the client device UI capabilities, every client device would have a Factory object. This object would have a global identifier and would provide a factory method for creating UI elements. Similarly, a Builder would encapsulate the UI creation logic for a particular UI technology. The UI technology would be identified by the UI Factory's global identifier.

JUnknown, the generic interface brings all the elements together by providing a method that dynamically loads the builder code from the service location for the particular client factory. Thus builder code is loaded dynamically from the service JVM to the client JVM. This is possible due to the underlying RMI support for such dynamic class loading.

4 Implementation details

The core classes in the framework are a part of the vrigo.core package. The generic interface that all jini services implement is called vrigo.core.JUnknown. The vrigo.core.Factory interface is implemented by all client factories and the service side builders implement the vrigo.core.Builder interface.

4.1 Classes

```
public interface JUnknown extends java.rmi.Remote {
    public Builder getBuilder(String factoryId) throws java.rmi.RemoteException;
}

public interface Builder {
    public void buildUI(Factory factory);
}

public interface Factory {
    public Object getObject(Object which);
    public String getID();
}
```

4.2 Sample Code

The client has a text based display. Its UI capabilities are encapsulated in a TextFactory class. The TextFactory is identified by the string identifier "vrigo.client.factory.TextFactory". After doing a jini lookup the client requests a TextBuilder from the service by passing the TextFactory identifier in the JUnknown's getBuilder remote method. The implementaion of this method is at the service end. The RadioService returns the TextBuilder which is loaded dynamically into the client JVM. A text UI construction can then takes place. Figure 1 shows the interactions.

```
// TextClient code snippet
JUnknown j = doJiniLookup(vrigo.core.JUnknown.class, serviceAttributes);
Factory f = new TextFactory(System.out, System.in);
Builder b = j.getBuilder(f.getID());
if (b != null) b.buildUI(f);

// RadioService code snippet
public class RadioServiceImpl extends UnicastRemoteObject
    implements JUnknown, RadioService {
    Builder getBuilder(String id) throws RemoteException {
        if (id.equals("vrigo.client.factory.TextFactory"))
            return new RadioTextBuilder();
    }
    other service specific code
}
```

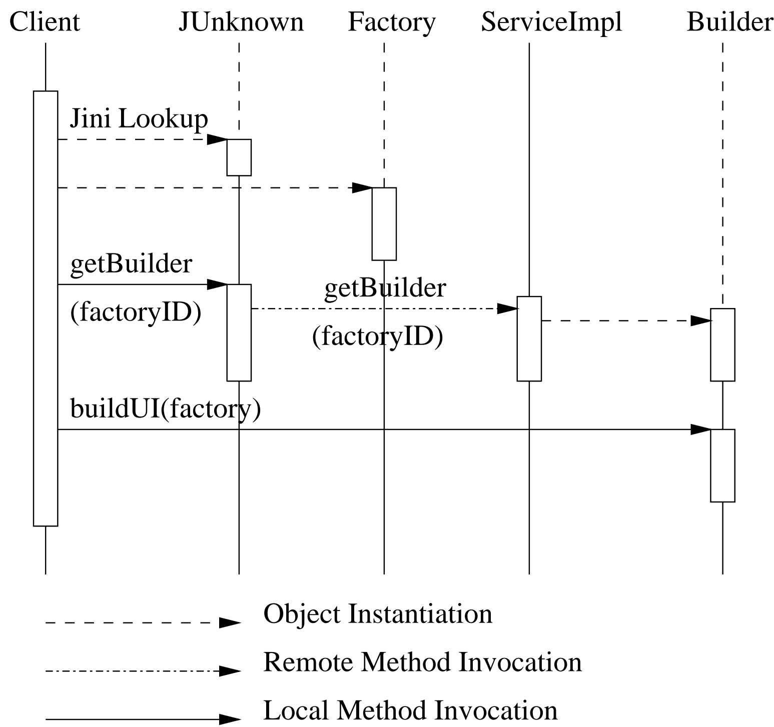

Figure 1: Interaction Diagram

5 Conclusion

Using the factory and builder patterns along with the JUnknown interface enables a client to do a serendipitous discovery of services in the local jini federation. Such services interact directly with the user utilizing the User Interface (UI) technology of the client device. This can enable a context aware computing environment where a user can control, communicate and compute-using the services around him.

It is instructive to apply this pattern based approach to enable non Java clients to talk to Jini services. We have implemented a WAPFactory and a WAPBuilder for a jini service enabling a user interaction from a WAP phone. This involves a Java application acting at a middle tier that does the jini lookup on behalf of the WAP phone. This application then dynamically loads the WAPBuilder and passes it a WAPFactory. The WAPBuilder outputs WML content to the WAP phone essentially enabling the user to control the jini service. To refine this concept and explore new application areas is our proposed future work.

References

[1] D. Siewiorek A. Mukherjee. Mobility: A medium for computation, communication and control. *In proc. IEEE workshop on Mobile Computing Systems and Applications*, 1994.

[2] E. Gamma et al. Design patterns-elements of reusable object oriented software. *Addison Wesley*, 1995.