Jini Connectivity for EIB Home and Building Networks: From Design to Implementation

Wolfgang Kastner, Christopher Krügel

Institut für Rechnergestützte Automation
Technische Universität Wien
Treitlstraße 1, A-1040 Wien
E-Mail: k@auto.tuwien.ac.at
URL: http://www.auto.tuwien.ac.at/jini

Abstract

After having launched Java back in 1995 Sun keeps up the hype by presenting Jini, a Java based middleware solution, a few months ago. Since this time a lot of articles have been published that describe scenarios where Jini enabled devices (mainly printers) provide virtually miraculous self-configuration abilities at a theoretical level. We want to go one step further by showing how a real Jini application can not only be thought of and designed but also be implemented. Because Jini is intended to be used in portable and household devices it seems natural to combine its power with another system often found in home automation, the EIB fieldbus. Our paper tries to show how these two systems can be linked together to benefit from each other.

1 Introduction

Java Intelligent Network Infrastructure (Jini) [1] is a concept that draws more and more attention among the ever-growing Java community. In most cases the future vision of linking a digital camera to a printer without having to configure a network connection is regarded as a challenge that Jini can address in an elegant way. However, there are much more capabilities of Jini than a mere decrease in network configuration complexity: Jini is also a good basis for integrating different services like telecommunication, consumer electronics or household appliances.

In a former paper [4] we explained how Jini can be used to extend the EIB system, i.e., the world of home and building automation systems, to various other services. In this context Jini – together with underlying control network technologies – could be the basis for the term “intelligent house” or “intelligent building”. Since the demand for increased comfort in homes and commercial buildings and the corresponding requirements of cost- and energy-saving will always fuel the use of modern networking technologies, these terms represent a very huge market!

This paper is structured as follows. We start with the introduction of fundamental Jini terms and their usage. In the following part we show the concept of an agent, a software module that represents a set of EIB devices at the Jini layer, and the pieces of information it needs to fulfill its task. After theses conceptual ideas we will present the most interesting implementation details and end with some conclusions and further research topics regarding this project.

2 Jini

Jini, developed by Sun Microsystems, is a distributed system that simplifies the access to services in a network. The outstanding advantages of a network are resource sharing and the possibility to use remote data and services from different points. It is a classical challenge for a network to provide lookup services for clients and to establish the corresponding connections dynamically. However, in most cases such connections have to be made by means of manual network configuration. The task of Jini is to simplify this configuration task by using refined protocols for lookup and other services. The easy usage of networked devices and services is one of Jini’s primary goal.

Because Jini extends Java, all Jini-based devices must feature a Java Virtual Machine (JVM), as indicated in Figure 1. The advantages of using Java as programming model are high reliability due to its strong typing system and automatic garbage collection. Furthermore Java supports the development of distributed applications by offering Remote Method Invocation (RMI) and a comfortable way of dynamic class loading.

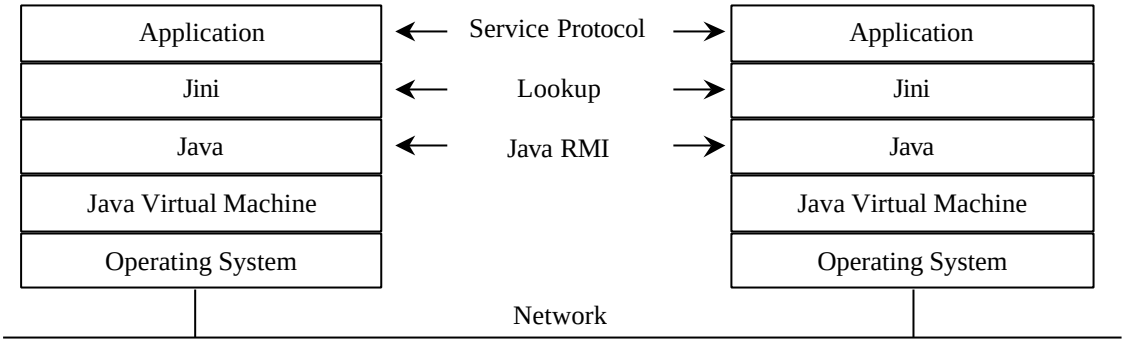

Figure 1: Jini Architecture

Jini, positioned in an area that is often referred to as middleware, is a logical layer built upon JVMs, that are connected over an arbitrary network. A set of Jini-based computing devices is called a Djinn. In a Djinn we find three different kinds of entities: services, service users and (at least one) lookup service.

- Service. A service is an abstract term for a resource that can be used via the network, like specific hardware (in our special case: sensors and/or actuators), software or a communication channel.
- Service user. A service user is the entity that uses one or more services. The protocol between service and service user is not defined at this stage, it is based on Java-based interfaces. Hence, the actual implementation of the communication between the two partners is defined by the service itself (notice: the underlying code is loaded dynamically by the service user!).
- Lookup service. Last but not least the lookup service acts as the middleman between services and service users.

In case someone wants to announce a new service at the Djinn, Jini provides the discovery protocol, that reveals all available lookup services. Afterwards the service may be registered by means of Jini's join protocol. If a service user is inclined to use a specific resource, the discovery protocol establishes once more a connection to an available lookup service. With the help of search attributes, the lookup service can find the services of interest. This is done by Jini's lookup protocol. Finally the service user can call the desired functions of the resource as indicated above.

In addition to this basic lookup functionality (which is suitable for dynamic adding and removing services to/from a Djinn), Jini also addresses networking issues like temporarily missing devices (e.g. due to a system crash) or privacy, authorization and authentication. The underlying concepts are protocols for leasing, transactions, security and events. Because a detailed discussion of all these Jini-related terms goes well beyond the scope of this paper we advise the interested reader to "look up" [2].

3 Design

The following design illustrates how Jini technology can be integrated into an EIB system. It's our aim to combine both systems such that

1. both of them can coexists and
2. devices of each system may interact with services of the other system.

Regarding our second goal EIB devices will be able to use services of the Djinn and - vice versa - Jini-capable devices will have access to the EIB system. Needless to say, that for the user's point of view, he/she will have the look and feel of a homogenous federation of devices.

3.1.1 Properties

In order to use services provided by an EIB device these services have to be equipped with certain properties, that (1) allow users to gain information about the type of the provided services, the abilities of the device as well as additional information that might be useful and (2) can easily be inserted into the Djinn. Hence, there are two major issues that have to be discussed: first it is important to know which information must be provided by each Jini-enabled EIB device to allow future clients efficient search mechanisms, second it is important how this information can be represented and stored (as described later in the section Implementation).

At the moment we discovered two major pieces of information important for each fieldbus device:

1. the functionality of the device and
2. the location
   a) of the device
   b) of the site where the device has been installed.

The first term stands for the services a device can offer and requires no further explanation. We admit that the other term holds two entities, they are tightly connected and can be bundled into one piece of information. However, why do we need data about the device as well as the exact location within the site? Two simple samples may answer this question:

1. When you want to switch off your lights in the living room with your Jini-enabled remote control, the corresponding Jini-enabled device must be found and can only be identified by its location within the house. Hence it is obvious important to know where a service can be found within a building.
2. When you are driving your Jini-enabled car and approach to your home, it is not sufficient for the car to look for garage lights because only the bulb in front of your garage should be turned on and not each one along your trail. Therefore it is also important to know the specific address of the site.

In our opinion the major different pieces of information are best kept in different data models, i.e., two separate trees, one representing the functionality the other one characterizing the location (Figure 2).

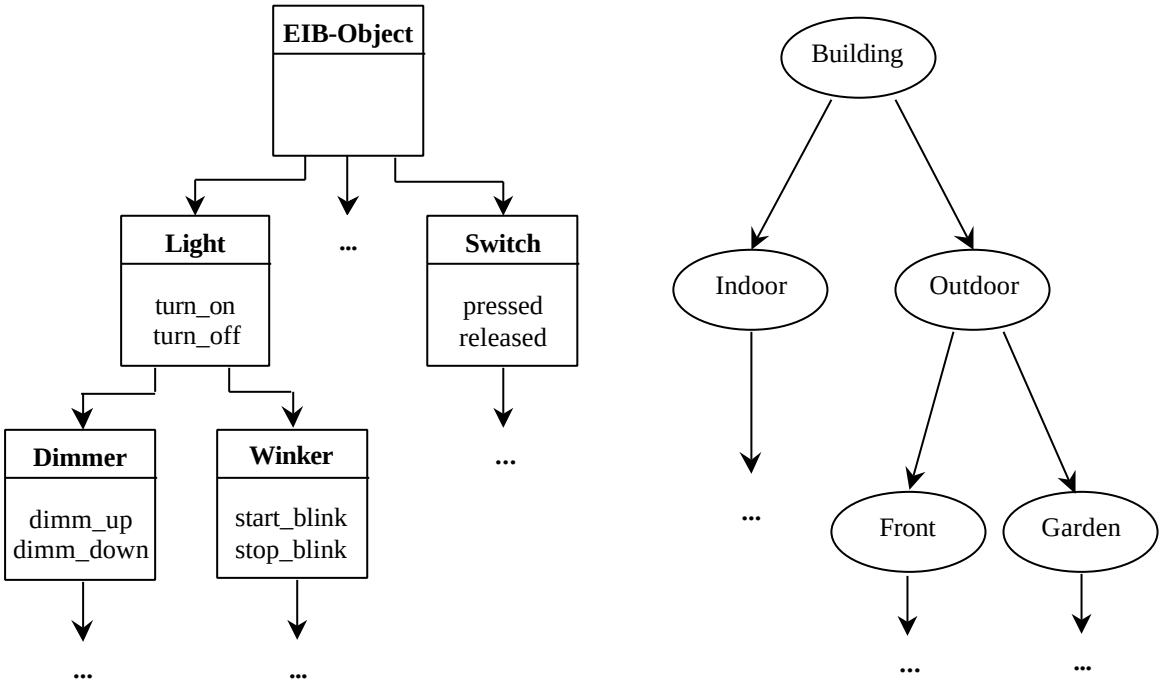

Figure 2: Functionality Tree and Location Tree

3.1.2 Agents

After this short introduction into the data model design, we may look behind the scenes of the system design. The main idea behind our EIB/Jini system architecture is to combine all logical parts of the EIB system (so called groups) into one agent, frankly called EIBAgent. Each EIBAgent consists of three parts (Figure 3).

1. Djinn Communication Module (DC) responsible
   (a) to find a lookup service (via the discovery protocol),
   (b) to announce the EIBAgent's services at the lookup service (via the join protocol) and
   (c) to search for services the EIBAgent wants to use (via the lookup protocol).
2. EIB Communication Module (EC) liable for the nitty-gritty fieldbus communication.
3. Jini Control Module (JC) acting as common control between DC and EC. After a new group of EIB-devices has been installed, the JC
   (a) configures the devices via the EC and
   (b) registers them at the Djinn via the DC.

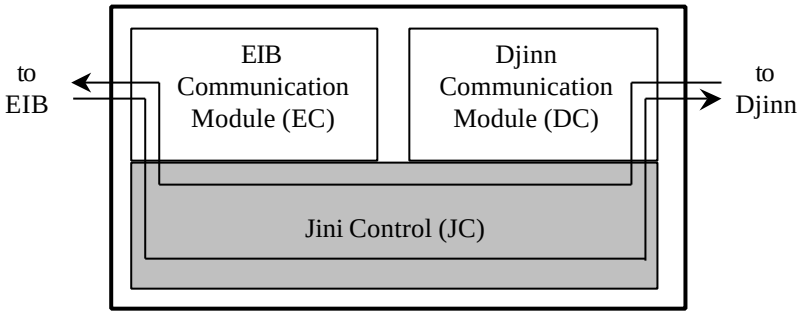

Figure 3: EIBAgent

Once an EIBAgent has been installed requests originating from the Djinn are subjected to the appropriate DC and forwarded by his JC to the EC. Requests coming from the EIB are subjected to the corresponding EC, handled by his JC and finally directed by the DC to the desired Jini-device. Since different Jini-capable EIB-devices may communicate through their EIBAgents, the further configuration of the EIB system (and if desired even the communication between “pure” EIB-devices) may be processed this way. Besides, communication between EIB-devices of different technologies (twisted pail, powerline) may be done by their EIBAgents. Hence low cost (software) gateways (e.g. twisted pair to powerline) are conceivable.

Consider once more a building equipped with EIB devices and a Jini-enabled garage light. A typical EIBAgent would be able to handle the lights of a certain room (LightAgent), another one would be able to control the heating of the room or even the whole building (HeatingAgent). Once installed the JCs will activate the related DCs, that in turn use Jini's protocols to register and join the Djinn. If a person enters the building, the light in the according room will be activated and the heating is turned on (latter of course only if it is too cold in the room).

Responsible for these actions are the DCs of the different EIBAgents. Incoming requests are passed through the corresponding JCs (perhaps with some additional security checks) to the ECs, that finally contact the fieldbus devices.

Since EIBAgents are modular and act autonomously, it is very easy to extend the system architecture presented so far to an EIB/Jini Manager (Figure 4).

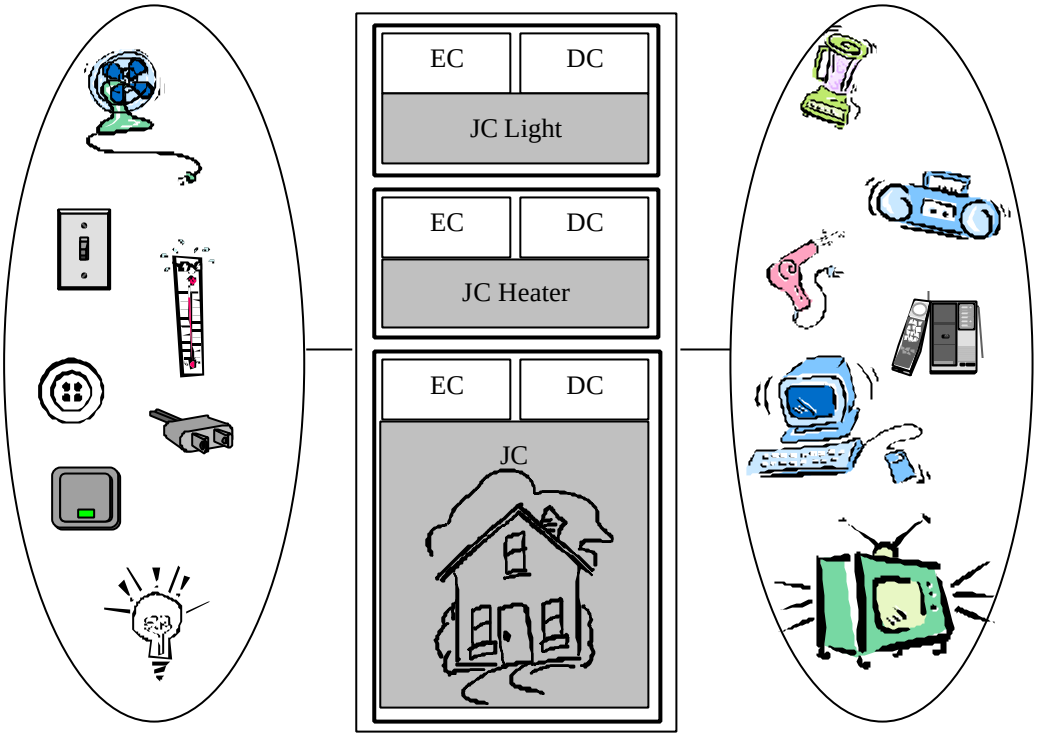

Figure 4: EIB/Jini System Architecture

EIB/Jini Manager could be responsible for whole buildings or even series of buildings. Keep in mind that

1. the integration of new EIB-devices in the building and
2. the (re-)organization of existing EIB-devices to new groups

can be done online adding newly created agents to an existing EIB/Jini Manager! To summarize, the following steps are necessary to integrate new agents into a Djinn:

1. install the EIB-devices into the EIB system
2. install the corresponding EIBAgent
   (a) extra EIB configuration is managed by the EC
   (b) Djinn-announcement is done by the DC

By means of these two information paths clients looking for specific EIBAgents can specify the desired services as well as their locations in a very comfortable and versatile way. Next we will show how this information is integrated into the Jini-based world.

4 Implementation

We are now ready to present some implementation issues based upon the above mentioned data and system architecture. For simplicity we use one characteristic scenario, i.e., our Jini-enabled car approaching the Jini-enabled outdoor light in front of our garage.

4.1 Properties

Because Jini is based on Java, an object oriented solution for our data representation seems natural. Jini offers two different ways to locate interesting services. One way is to search for a certain class or its derivates, the other one uses additional classes (entries) to provide information about certain fea-

tures. According to our data model our solution uses both mechanisms to provide clients an easy way to connect to the desired EIBAgents.

The functionality (describing the services an EIBAgent can offer) is best represented by different Java method signatures that can be called by clients. While it would be possible to describe functionality in other ways (e.g. by a string representation), it seems natural to map services directly to their function calls with the description of the needed arguments to call the service.

As presented in the last section we use a functionality tree to group all possible services of EIB devices. Derived from an abstract top level EIB device, each class of devices is inserted into the functionality tree by extending a parent class which offers some similar services and adding additional methods for special unique services. An example for a part of the functionality class tree is given below, where the top class EIB device (cf. lines 00-03) is extended by different light services (cf. lines 05-08 and 10-13) that offer more and more special operations.

```
00: public class EIBObject {
01:         String manufacturer;
02:         long VersionNumber;
03: }
04:
05: public class Light extends EIBObject {
06:         void turn_on()      {}
07:         void turn_off()     {}
08: }
09:
10: public class Dimmer extends Light {
11:         void dimm_up()      {}
12:         void dimm_down()    {}
13: }
```

While it could be desirable to allow a new device to extend more than one parent and thereby inheriting operations of different classes (e.g. a dimmable and blinking light could extend a dimmer and a winker class) this is prevented by the simple inheritance model of Java. Nevertheless this drawback is more than outweighted with the advantage of using the functionality tree directly in Jini for the class base search.

When clients look for a certain kind of functionality they just specify the desired class and the lookup service returns all objects that implement the needed service (and probably much more, as all derived classes are automatically selected as well). Keep in mind, that when functionality is represented with the help of interfaces (thereby allowing multiple inheritance), it cannot be guaranteed that the returned objects really implement the desired service and that might result in class cast exceptions.

In our approach such problems can never occur, since the functionality is represented by the type of the class the agent is member of! However we admit, that it is obvious that in order to be able to use this model on an industrial level, manufactures of EIBAgents have to agree upon a common functionality tree. Nevertheless it is easy to use common structure describing languages (like XML) to build this functionality tree and then convert this information into Java classes automatically!

To refine a search for a specific service Jini offers additional classes (so called entries) that provide information about certain features. We use these entries for storing the address information – conform to our data model – now building up a second hierarchical tree, called the location tree. Again, this data structure has several advantages: for instance, consider a group of EIB devices spread over a couple of different places. It would be inappropriate to equip the EIBAgent with different entries representing each room. Much better is to equip it with one single entry representing the whole group of rooms.

Typical location trees consist of a common root (e.g. the building) divided into some major regions that are further divided into several minor regions. The address itself is represented as a string attribute of each location class. This way the location tree can easily be used in the entry based search during Jini lookup process. An example for a part of a location class tree is given below, where the top class (cf. lines 00-08) is extended by different regions (cf. lines 10 and 12-16).

```
00: public class BuildingEntry implements Entry {
01:         String location;
02:         Orderid id;
03:
04:         BuildingEntry(String s) {
05:                 location = s;
06:                 generateOrderid(id);
07:         }
08: }
09:
10: public class OutdoorEntry extends BuildingEntry { ... }
11:
12: public class GarageEntry extends OutdoorEntry {
13:         GarageEntry(String s) {
14:                 super(s);
15:         }
16: }
```

To illustrate once more the way a client can search for different services and more agents than one in a single lookup we imagine a house with a garage light and a light at the front door that is capable of blinking. When the house owner comes home his car starts a lookup for light/garage services and turns on the light. When he leaves his house through the front door and it is dark outside the door sensor starts a lookup for light/external services and finds both lights (front door and garage) and turns on both of them. When the burglary alarm system is activated it searches for blinking-light/building services and all light sources inside and outside the house that are able to blink will do so.

4.2 Agents

Last but not least we will show the most interesting program parts of one EIBAgent. Again we use the light agent as an example. The code for the light agent can be stored in one file, for instance called `LightAgentLauncher.java`. Inside this file we find two classes. The public class `LigthAgentLauncher` holds part of the JC code and an inner class `DjinnControl` (for the DC code). The second class `LightAgentProxy` represents the code that is moved over the network and consists of parts of the JC code and an inner class `EIBControl` (holding the EC code).

Before Jini-enabled devices can take advantage of our Jini-enabled light, the launcher has to find at least one Jini community (i.e., the Djinn). As already mentioned the process of finding available lookup services is called discovery. Afterwards the EIBAgent is able to join all lookup services returned from the discovery process, actually done via the join protocol. Hence, the basic steps our LightAgentLauncher has to do are

1. find the lookup services that may be active (cf. line 21). Whenever a new lookup service is found our object `lookup` (instance of the Jini class `LookupDiscovery)` calls out its registered listeners. To add our specific listener `DjinnControl` we simply use the method `addDiscoveryListener` (cf. line 22).
2. integrate the service into the Djinn, that means
   - create a `ServiceItem` that holds the proxy object for the service (i.e., `LightAgentProxy`) and attributes that describe it (cf. line 20),
   - publish the service by calling `register()` on any `ServiceRegistrars` found through the discovery process. This obviously is done via our listener `DjinnControl` (cf. line 11) and
   - maintain the leases on service registrations returned form the lookup services (cf. line 24).

Once downloaded the LightAgentProxy performs the service requested (cf. lines 39-42 and 43-46). Strictly speaking whenever a Jini-enabled device finds our LightAgent via the lookup protocol and the queried service items (derivable from the location tree), the proxy object will be copied into its JVM. Afterwards the device can use methods read out of the functionality tree to use the service.

However, to download the proxy object to the remote JVM the object has to be "serializable" (cf. line 27). As soon as the proxy has been transferred, the downloaded object may use a private communication protocol for talking to the specific device (cf. line 32). In fact in the actual implementation we use TCP/IP sockets to talk to an EIBServer that has access to the EIB system via FALCON [6].

```
00: public class LightAgentLauncher {
01:       ServiceItem item = null;
02:       ServiceRegistrar[] allFound;
03:
04:       class DjinnControl implements DiscoveryListener {
05:             ServiceRegistrar[] newregs = null;
06:             final long leaseTime = LEASE_TIME;
07:
08:             public void discovered(DiscoveryEvent ev) {
09:                   newregs = ev.getRegistrars();
10:                   for (int i=0; i<newregs.length; ++i)
11:                         newregs[i].register(item, leaseTime);
12:                   addRegsToAllFound(newregs);
13:             }
14:             public void discarded(DiscoveryEvent ev) {...}
15:       }
16:       public static void main(String args[]) {
17:             LookupDiscovery lookup;
18:             Entry[] e = new Entry[1];
19:             e[0] = new GarageEntry("Karlsplatz");
20:             item = new ServiceItem(null,
                        new LightAgentProxy(StringToGroupAddr(args), e);
21:             lookup = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
22:             lookup.addDiscoveryListener(new DjinnControl());
23:             while (true)
24:                   renewLeases(allFound);
25:       }
26: }
27: class LightAgentProxy extends Light implements Serializable {
28:       EIBControl ec = new EIBControl();
29:       GroupAddresses groups = null;
30:
31:       class EIBControl {
32:          send_data_to_bus(Data data, GroupAddress ga) {
33:          // use proprietary protocol to send data to
34:          // groupAddresses on fieldbus
35:      }
36:       LightAgentProxy(String[] address) {
37:             groups = new GroupAddresses(address);
38:       }
39:       void turn_on() {
40:             ec.send_data_to_bus(new Data("turn_on"),
41:                               selectGroupAddress(groups));
42:       }
43:       void turn_off() {
44:             ec.send_data_to_bus(new Data("turn_off"),
45:                               selectGroupAddress(groups));
46:       }
47: }
```

5 Conclusion and further research

This paper explains how certain services of an EIB system (e.g. lights/luminaires, heating system, hot water, alarm installation and many more) can be integrated into a Jini environment. Since the infrastructure of a typical EIB installation does not fully comply with the environmental assumptions of Jini, the paper discusses the use of a proxy-oriented system architecture. After a short introduction to Jini-technology, we showed the design and implementation how to Jini-enable EIB devices. Following

our approach the interaction between EIB and Jini will lead to a new definition of the term "control network" itself. This approach follows our credo read "**The Network is the Device!**".

The system presented so far works fine when the EIB system is initially configured and all EIBAgents get the group addresses of their managed devices at instantiation time (cf. line 20 in the LightAgentLauncher's code). Unfortunately problems occur when group addresses are changed or when a device is removed. When a certain group address is no longer used by a device all telegrams that are sent to it by its agent are lost when the agent is not informed about the changes. In our system all devices are unaware of their agents and cannot inform these agents by themselves. As a result the information flow from fieldbus devices to their agents and to the Djinn poses an additional problem. When EIB devices are not aware of their managing agents, how do their telegrams reach the appropriate agents?

It is not desirable to implement the forwarding mechanism in the service process that manages the bus access itself because this server should only be used as a gateway to translate between local network and fieldbus communication. The solution to these problems is the introduction of a database that manages the correspondence of EIBAgents and their methods (services) on one hand and the group addresses of the managed devices on the other hand. This newly inserted database is consulted each time a method of an agent is invoked by a service user within the Djinn and returns a set of group addresses where telegrams have to be sent to. The database is used for information flow in the other direction as well because for each received telegram from the bus all interested agents can be retrieved and the information can be forwarded. Each time a method of an EIBAgent is invoked by a Djinn service user it contacts the JDBS (Jini Database Service) to get the corresponding group address(es) to contact the fieldbus. Telegrams from the bus are sent to the JDBS which then consults its database and forwards the messages appropriate. Notice that the JDBS could itself be a Jini Service and its services could be made usable networkwide. At the moment this database information cannot be extracted directly from the EIB database created by means of the ETS. Hence we either intend to use EAGLE for accessing the EIB database or (later on) try to integrate our JDBS into the ETS, for instance via plugins.

Another problem we are faced with is the fieldbus access. In the current version we use FALCON and therefore Microsoft specific extensions (i.e., COM accesses [5]). That's why we have to run our prototype implementation under a Microsoft-JVM. However, one of our next steps will be to implement a Linux device drive to solve this shortcoming and provide multiplatform support.

Soon a lot of exciting research and development work has to be done. The interested reader should consult our homepage http://www.auto.tuwien.ac.at/jini where ongoing work and future results will be presented

Acknowledgements

We would like to thank the European Installation Bus Association (EIBA) and Asea Brown Boveri (ABB) for sponsoring this research project. We explicitly acknowledge our former colleague Dr. Heinrich Reiter (now member of the EIBA) for useful Jini discussions and his logistic support since start of the project.

References

[1] Sun Microsystems Inc.: *Jini Architecture Specification*, http://www.sun.com/jini
[2] Edwards K.: *Core Jini*, Sun Microsystems Press, 1999.
[3] European Installation Bus Association: *System Introduction*, http://www.eiba.com
[4] Kastner W., Krügel C., Reiter H.: *Jini ein guter Geist für die Gebäudesystemtechnik*, Proceedings JIT'99, Springer-Verlag 1999, pp. 213-223.
[5] Microsoft Inc.: *Java and Com*, Microsoft SDK for Java, 1999.
[6] Sahm C.: *Falcon on-line Help*, Falcon Windows-Helpfile, 1999.