IMPROVED FIELDBUS CONTROL VIA MIDDLEWARE TECHNOLOGY

W. Kastner, C. Krügel

Technische Universität Wien
Institut für Rechnergestützte Automation
Treitlstraße 1, A-1040 Vienna, Austria
Email: k@auto.tuwien.ac.at

Abstract: After having launched Java back in 1995, Sun keeps up the hype by presenting Jini, a Java based middleware solution, a couple of months ago. Since then a lot of theoretical work has been published, which describes scenarios where Jini enabled devices provide virtually miraculous self-configuration abilities. This paper goes one step further by showing how a real Jini application cannot only be thought of, but also be designed and implemented. Since Jini is intended to be used in portable and household devices, it seems natural to combine its power with a fieldbus system for home automation. This paper presents how Jini and a home and building network can be linked together to benefit from each other. *Copyright 2000 IFAC*

Keywords: fieldbus, home and building network, spontaneous network

1. INTRODUCTION

To gain a better understanding of the motivation and the design of an integrated environment between Jini and a home and building network, the article starts with a short description of both technologies.

1.1 *Jini*

Jini, developed by Sun Microsystems, is a distributed system environment that simplifies the access to services in a network (Edwards 1999). The outstanding advantages of a network include resource sharing and the possibility to use remote data and services from different points. It is a classical challenge for network management to provide lookup services for clients to establish the corresponding connections to a server dynamically. However, in most cases such connections have to be made by means of manual configuration.

The task of Jini is to simplify this configuration job by using refined protocols for lookup and other services. Thus, the easy usage of networked devices and services is one of Jini's primary goal. Simple devices are able to spontaneously form communities, called Jini federations, where they utilize each other's services. An architectural overview is presented in Figure 1, where the cooperation between the application, Jini and the underlying Java layer is shown.

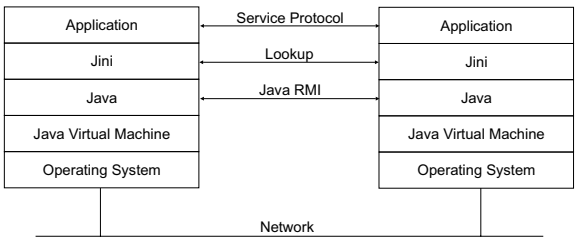

Fig. 1. Jini Architecture

Jini is an extension of Java, so all Jini-based devices must feature a Java Virtual Machine (JVM), as indicated above. 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 automatic class loading on demand.

1.2 *Home and Building Networks – the EIB*

Fieldbus and installation bus systems can be used in a wide area of different applications. While they were mainly used in technical control and process automation settings for computer aided manufacturing and for the automotive branch, they get more common in home and building automation to control simple actuators like lights or sensors like switches.

The European Installation Bus system (EIB) for home and building electronics was designed to answer the last mentioned challenge (EIBA 2000). In contrast to the island solutions of conventional electrical installation, EIB unifies all aspects of building exploitation on a shared communication medium. An EIB control network is marked by a clear structure aimed for home and building automation, permitting a line, tree or star topology. In this way, EIB components (or devices) are grouped into lines, up to 15 of which may be connected to a main line through line couplers (with routing functionality) thus forming an area. In turn, a backbone line may link up to 15 areas. As each line accommodates maximally 256 devices, a fully equipped EIB system may link roughly 60000 devices. This topology (or a subset) may be realized on twisted pair cabling, or alternatively using power-line or wireless radio-frequency transmission.

Each EIB device contains a microprocessor. It implements the stack for decentralized EIB communication and provides processor, memory and communication facilities to the local application. Certain standard building blocks (Bus Coupling Units) offer a well-defined interface to the application-specific hardware such as sensors or actuators. Devices may be managed individually across the network through physical addressing. For their runtime operation, applications typically communicate via multicast-enabled process variables, meaning they may be logically grouped and considered as a single shared variable. These links themselves may also be assigned or modified over the EIB network.

The EIB allows a user to perform simple tasks like lighting and heating control, but also makes it easy to control air-conditioning or power management processes and joins them into an integrated system ("Intelligent House"). Although the EIB calls itself an installation bus, it is actually more an instance of a fieldbus system, capable of solving challenging control applications.

The rest of this paper is structured as follows. After a short motivation why an integration of home and building appliances with Jini seems useful, the system design for such a connection is presented. Section 4 summarizes the most important topics of the implementation. Some conclusions of the work done so far and an outlook on future research regarding this area finish the article.

2. MOTIVATION

When considering to merge the EIB with Jini, it is interesting to compare the different motivations behind each system and to find out, whether these motivations or architectures can be connected to gain an increased benefit.

Jini is a new concept, that provides the fundament for spontaneous connection of devices to form temporary networks. Devices are gathered to solve a common task and eventually leave the federation (while forming a new connection to reach a different goal). In contrast to that, the EIB is a long lasting and static installation. EIB devices are connected to a common bus, that is likely to be wired inside the walls of a building. They get programmed once and then fulfill their tasks by sending telegrams to interested partners when state changes occur or receive a telegram when they should perform their outputs. At a first glance, there seems no need for EIB gadgets to form changing subnetworks.

Looking closer reveals, that, if federations including bus devices are opened, Jini can suddenly become useful. The EIB devices are mapped onto services in a Jini federation, that are not only used by other bus installations but raised to an open environment, where other devices may make use of the fieldbus. One one hand, it now becomes possible for other Jini enabled devices to use fieldbus gadgets and control actuators, as well as receive information from sensors to improve their knowledge of the surrounding. On the other hand, it becomes possible for EIB devices to extend their functionality by using other installations outside the regular bus installation. Kastner *et al.* (1999) lists some examples for possible functionality enhancements like

- The remote control of the TV set can be used to regulate the heating or control the venetian blinds. The user interface is shown via video text and responses of the EIB devices are visualized on the screen.
- When the TV set is turned on, the lights of the appropriate room are dimmed or turned off, depending on user's settings.
- When a person enters the building (or parts of it, depending on the size of the house)

the heating or air conditioning automatically turns on (depending on temperature values, EIB devices have recorded).

The increases in functionality mainly result from a connection between consumer electronics and home automation devices produced for the EIB. While it would be possible to create proprietary links between single devices, that could perform the same (e.g. implement an EIB heat sensor into the oven and connect it with EIB lights), the solution that uses Jini has the advantage of being vendor independent and allows the manufacturer of a special device to equip it with functionality, that works in an open environment.

Additionally, a couple of other benefits immediately arise when using Jini compared to a proprietary fieldbus-to-consumer-electronics connection. These are

- a gateway from the fieldbus to the Internet,
- the possibility to provide facility management,
- a form of ubiquitous computing and
- a cheap gateway between different fieldbus variants (like EIB, LON or CAN).

3. SYSTEM DESIGN

Jini assumes a complete Java environment to run properly. This is no problem, when a standard PC, that runs a fully-fledged JVM, is used as a platform. Nevertheless, these demandings become much more restrictive, when we think about small devices. When a simple actuator (e.g. a light) or sensor (e.g. heat sensor) should act as a Jini service, the scarce resources become a problem. It seems evident, that it is not useful to provide each component with enough computational power and memory to run Java on its own. In Sun (2000) three different solutions have been proposed, which reduce the flexibility by decreasing the needed resources.

(1) Use a proxy with a complete JVM to act on behalf of the simple devices. This variant needs a shared virtual machine, that runs Jini protocols and performs communication on behalf of its clients.
(2) Implement a reduced JVM in every device, that is just able to announce and provide its service. This is mainly an option for service providers, as clients need to deal with unknown code.
(3) Implement a complete JVM with a reduced need of resources by utilizing techniques like dynamic compilation, special purpose Java processors or automatic library down-loading. These techniques mainly combat Java's deficiencies like indeterministic garbage collection or weak run-time performance, issues especially severe in real-time applications.

Our architecture uses the first variant, a gateway solution with a shared virtual machine running on a workstation (PC), that hosts service proxies on behalf of their simple clients (Kastner and Krügel 2000). The shared JVM acts as a pool for software modules, that correspond to a set of devices on the EIB and offer the devices' functionality. These software modules are called *agents* (or EIBAgents) and act as proxies for EIB devices in the Jini federation, which they have to represent.

An agent is more a representative of a certain fieldbus functionality rather than an avatar for a special device. An example could be an alarm agent, that periodically turns on and off the outdoor lights and rings a bell. In that case, a single software module has to handle a set of EIB lights and a bell actuator. As a result for our design decisions, it is important to notice, that there is no one-to-one mapping between fieldbus devices and software modules.

Obviously, it should also possible to have a single fieldbus component controlled by more than one agent. This might be useful when a certain device offers different functionality (e.g. a switch, that also features a LCD to display short strings) or is needed for different tasks (e.g. a light that can be turned on and off by a simple light agent and by an alarm agent). The system presented here is capable of an unrestricted n-to-m mapping between agents and EIB devices. Since the agent concept is the centerpiece of the whole design, the structure of a single agent is described below in more detail.

3.1 *Agent Design*

An agent needs to provide the EIB device functionality as a service in the Jini federation. Service requests from other Jini participants have to be forwarded to the fieldbus, demands of fieldbus gadgets have to cause service calls in appropriate Jini servers. Next, it is necessary to inform the agent of state changes of devices located at fieldbus level. To allow an agent to manage its tasks, it must be able to communicate with the Jini federation as well as with the fieldbus. Because telegrams can be sent in both directions, an agent needs the ability to read from and write to the fieldbus and to communicate bi-directional with other Jini services. As a last piece, a part of the agent has to realize the demanded functionality. This can be as simple as just forwarding a request to turn on a light to the corresponding light de-

vice, but become as complex as the control and monitoring of an air conditioning system.

Following object-oriented design principles, an agent is splitted into three parts (Figure 2):

(1) Djinn Communication Module (DCM) – responsible for communication with the Jini federation.
(2) EIB Communication Module (ECM) – responsible for communication with the fieldbus (EIB).
(3) Jini Control Module (JCM) – responsible for bootstrapping and agent functionality.

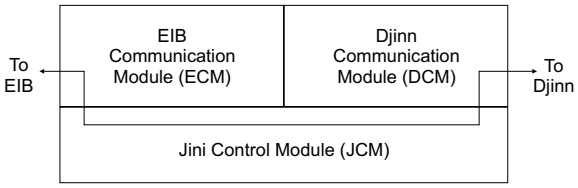

Fig. 2. Agent Structure

Our gateway uses a serial connection to a twisted-pair bus, which demands special considerations when providing an implementation for the ECM. Agents are Jini proxies, that may move around in the network. It is therefore not possible to guarantee, that the ECM will be located at the host with the fieldbus directly connected to its serial interface. While this seems to be an implementation detail at first, this is not true. The design must add an additional layer of indirection between the actual fieldbus connection (via the local interface device driver) and the ECMs, that can be located anywhere around the network. Therefore, it is necessary to provide a server process, which is frankly called *EIBServer*.

The EIBServer runs locally on the machine, which provides the connection to the fieldbus. The ECM gets relieved from the need of implementing different communication patterns for different fieldbus media types, because these differences are hidden behind the EIBServer. The duty of the ECM is reduced to the need of providing a location-transparent connection between the EIBServer and the EIBAgent, namely the JCM.

3.2 *Agent Properties*

An agent is registered at the Jini federation by the JCM. The JCM uses a special protocol (called join) to register a service object at the *lookup service* (LS). The lookup service itself is a Jini entity, comparable to a name service, that can store a mapping between object descriptions and the objects themselves. It is evident, that the functionality and the interoperability of the system depends on the ease, other Jini services can find and use EIBAgents. To allow users to look up services quickly, they are equipped with certain properties (called entries in Jini). This section deals with the properties of agents and mechanisms to provide a safe and fast way for clients to utilize the LS and get the desired service proxies.

It is obvious, that a service user, who needs an agent to fulfill his task, has a clear idea of what the device needs to do. Jini uses two different ways for service users to specify a proxy object. One way is to specify the type of the service class or interface, the other way is to provide entries (i.e., additional classes), that act as service descriptions.

The functionality of an agent is defined by all procedures, that a client may call. Thus, it seems a natural approach to code the agent's functionality by the class or interface types, it has to implement. This approach allows to make use of Java's inheritance mechanism by creating generic classes or interfaces with a basic functionality (e.g. a light interface, that only has procedures to turn on and off the light) and sub-classing them by proxies, which provide additional services.

Starting from an abstract interface called *EIBObject*, that represents an arbitrary EIB device, a tree can be formed. Interfaces, that are near the root, only offer some high-level functions for a certain home automation area (like heating or lighting) and have sub-nodes, that augment these basic services with special, possibly vendor specific extensions. As an example, a small object tree with some lighting interfaces is shown in Figure 3.

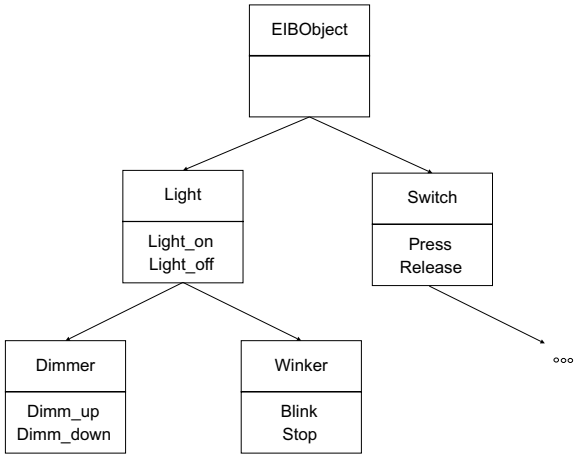

Fig. 3. Agent Functionality – Class Hierarchy

Compared to a variant where the functionality is coded into additional entries, our mechanism offers two advantages. First, the polymorphism of the Java object model can take effect, when clients issue their search. A service user, that is looking only for basic services, also receives proxies with an increased amount of functionality, if they extend the class offering the basic functions. The second advantage is the increase in safety when mak-

ing a function call. A proxy object, that is received by the lookup service, has to be down-casted to the needed object in order to call its procedures. When functionality is represented by entries, it is possible, that a service object pretends to offer certain procedures, which its class actually does not. In that case, the downcast would result in a Java exception and needs special error treatment at the agents. This problem cannot occur when service functionality is directly mirrored by the callable functions and the object type.

Another important aspect of a device, that has to be reflected in an agent, is its location. It is important for many applications to select certain devices according to their functionality and their location. The information of the place is kept where the device is installed as an entry (storing them in a hierarchical manner, too).

3.3 *Agent Proxy*

Whenever a service registers itself with the lookup service, a service proxy object has to be sent to the LS. This service proxy is eventually transfered to clients, which in turn can use it to perform service calls. In case of an fieldbus agent, it is possible that multiple clients throughout the network simultaneously hold a reference to the agent's proxy object. Whenever an event on the fieldbus takes place, all agents have to be informed about the state change consistently, regardless of their current location within the Jini network. Nevertheless, it is not desirable that a pressed switch on the EIB causes all agents to react. These telegrams would contain redundant information and simply overload the bus. An ideal situation would be an agent, which runs as a single instance, when bus traffic is received, but can run as many copies of itself as needed, when other Jini services want to access the fieldbus via the agent. Although this dual role seems to be unachievable, the use of RMI makes it possible.

Remote objects, which are passed as parameters in RMI function calls, do not copy the objects itself but only a remote reference (called stub). These stubs may be passed around the network to clients, while only a single instance of the agent is actually running on the host, where the agent has been started. Defining an agent proxy as an RMI server object has the additional benefit of allowing the proxy object to act as a remote event listener as well. Jini supports a remote event mechanism, a way to inform applications about the occurrence of asynchronous events. As agents are already build as remote server objects, it only needs minimal additional effort to allow them to directly receive events, which indicate state changes at EIB devices.

4. IMPLEMENTATION

The notion of an agent has been described as the centerpiece of the system design. Agents act autonomously in the Jini federation and the system can easily be extended by integrating new agents into the community. It is interesting to notice, that large parts of the code are similar between all agents. The routines to contact the lookup service and publish its service there as well as the fieldbus communication are the same for all agents. Only the small amount, that defines the actual abilities of the agent, needs to be rewritten for different fieldbus devices.

However, agents are not sufficient to build up the whole system. It has been shown, that it is necessary to insert an EIBServer, a layer that manages the connection to the fieldbus via the local device driver on the gateway host. Another challenge is the communication between the agents and the EIBServer, which is handled by the ECM on the agent's side. The server side is more complex to implement, as it is necessary to allow a single packet from the fieldbus to be forwarded to more than one agent and cause an asynchronous notification there. The information of the mapping between a single EIB device and the set of agents has to be stored at the EIBServer and is implemented as a database.

Figure 4 shows the whole system architecture with the EIBServer divided into the two modules, called *JiniDatabase* and *EIBController*.

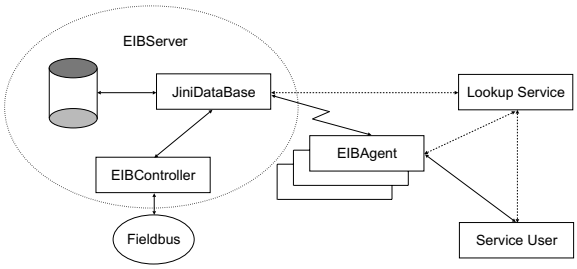

Fig. 4. System Architecture Overview

The connection between the JiniDatabase and agents is shown as a lightning, that represents the asynchronous notification by remote events. The dashed lines represent Jini communication with the lookup service and solid lines show regular function calls over the network (via RMI or sockets). This diagram presents an overview of the design and highlights the position of agents in the whole system.

The JiniDatabase has been introduced to insert an additional level of indirection between agents and fieldbus devices, latter being represented by their group addresses. An agent is given a unique identifier, which is used to index a database, where all corresponding group addresses are stored. This

database is used for input from the fieldbus as well as output to it. Different connections for both data-flow directions can be established.

The concept of introducing an additional layer to connect agents and their managed devices has two advantages compared to a model, where every agent stores its own group addresses. First, a performance advantage is gained, when a proxy object has to manage a big set of devices. Instead of transmitting all group addresses from agents to the database for every access, only a single identifier must be sent. Second, the database allows tools to change the connection between devices and agents without the need of informing or restarting all proxy objects on the network. It is possible to reconfigure or add/delete new devices with their addresses remotely, without interfering with running agents. Additionally, the Jini Database provides a convenient way of remembering, which agents need to be informed, when a bus telegram is received by the EIBController. Without the database, the connection between group addresses and agents, that need to be informed about state changes, would have to be managed somehow by the EIBController itself. This could be done by hard-coding links or reading configuration files – both variants, which cannot keep up in flexibility with a database solution.

Our test environment consists of various EIB devices (binary inputs/outputs, dimmer outputs, switches equipped with LC displays), light bulbs and regular switches. Since no Jini-enabled physical devices, which could be used to test the shown software solution, were at our disposal, it was necessary to write a simulation environment, that implements some functionality offered by such devices. Hence, a graphical interface was designed, which allows access to a number of virtual devices, that provide some Jini functionality.

The machine, that is hosting the shared JVM and providing the connection to the EIB via a serial port, is a standard desktop PC (Pentium II-450, 256 MB RAM). The EIBController needs a Microsoft Java virtual machine, so MS SDK 3.2 is used together with the standard Java Development Kit (JDK) 1.2 by Sun, that is utilized to compile and run all other classes. The JVM used has a common garbage collector working in the so-called stop-the-world mode. That means, that the collector is only called, when the available memory is exhausted and an allocation fails. Notice, that it is no problem in our application area (i.e., since the execution of the application is suspended for a couple of seconds, a command forwarded to the EIB could last longer), but this could be disastrous for real-time systems, which have to meet hard deadlines.

The whole project (i.e., design and implementation) was done in approximately 4 months. To extend the existing software to a real EIB installation should not be a problem, since the additional programming effort to be done is to integrate new agents into our well defined class hierarchy.

5. CONCLUSION AND FUTURE WORK

This paper described the implementation of a home automation system using the EIB fieldbus as a fundament for controlling actuators as well as sensors and extending its functionality by means of a middleware layer.

The presented system itself is designed for larger installations, where a dedicated personal computer is used to realize the additional tasks and serves as a Jini platform. But it might be too expensive to purchase an extra PC for every simple EIB facility in private homes. Here, our work offers a starting-point for further research. It might be desirable to implement an embedded Jini Controller, that allows to keep the needed Jini functionality at every device, simply by having every fieldbus gadget equipped with a JVM, that hosts the Jini classes itself.

In comparison to the proxy solution, where the fieldbus is simply accessed by a network driver and every device needs a certain representative, called agent, that is distributed in a separate network (local or even the Internet), the implementation of a Jini Controller changes the fieldbus installation itself. The agents are inherently distributed, by having exactly one running at each device. It may still be possible to combine certain gadgets to groups and having them controlled by a master fieldbus device, but the general idea is to have each device represent itself as a Jini service. Instead of having a fieldbus and a local network (e.g. Ethernet), that run different protocols, both networks can be integrated into a single installation.

REFERENCES

Edwards, K. (1999). *Core Jini*. Prentice Hall.

EIBA (2000). European installation bus system introduction. *http://www.eiba.com*.

Kastner, W. and C. Krügel (2000). Eib and Jini – from concept to implemenation. *EIB-Proceedings* **3**, pp. 41–55.

Kastner, W., C. Krügel and H. Reiter (1999). Jini: ein guter Geist für die Gebäudesystemtechnik. *Proceedings JIT'99* pp. 213–223.

Sun (2000). Jini: Java intelligent network interface. *http://www.sun.com/jini*.