Dynamic Device Management and Access based on Jini and CAN

Dipl.-Inform. Gerd Nusser[1,2] Prof. Dr.-Ing.Gerhard Gruhler[2,3]

[1] Wilhelm-Schickard-Institute for Computer Science (WSI), University of Tübingen
[2] Institute for Applied Research (IFA), University of Applied Sciences Reutlingen
[3] Steinbeis-Transfercenter Automation (STA), Reutlingen

In this paper, we introduce an object-oriented client-server system which is primarily based on Jini and CAN, where Jini is used as a middleware architecture which offers devices and their services to a network, such as the internet. The dynamic integration of services and the feasibility of network pluggable devices are the major benefits of using Jini. As Jini offers the ability to move code over a network, it is possible to use device-specific application code and device specific options dynamically. The application to access a specific device is transparently transferred to the client.
For demonstration purposes a prototype to access CANopen devices over Jini was developed. The client, a PDA (personal digital assistant) with a small embedded Java Virtual Machine is used to access CAN devices connected to an embedded Java system called TINI. As nowadays no devices directly supporting CAN and Jini are available, so called proxies are used to tie up the systems to the Jini community.

1. Introduction

The availability of network-based systems, like the Controller Area Network (CAN), have changed traditional automation systems in a variety of ways. In the past, these kind of networks were closed entities, which in some cases is inevitable (e.g. in mission or life critical systems), but the need for serviceability and therefore an external interface gets more and more important. It is necessary to maintain systems locally as well as over a long distance. Therefore the need for an open service and management infrastructure, by means of diagnostics, is immanent.
Additionally, the way we think about systems will change and "the number of actively and co-operatively interacting parties in the internet such as traditional increase even more" [2]. Furthermore, the need for integrating systems in the overall infrastructure of a company will be as much important as managing and accessing these systems over a large distance where the Internet can serve as "an appropriate transfer medium" [1]. To interface traditional systems like CAN to the internet we need a dedicated instance (a gateway) like a personal computer as described in [3] or an embedded device with CAN and an Ethernet interface. The fact that devices with an additional "Ethernet interface results in higher costs" still holds, but gets less expensive like the embedded system shows, which is presented in section 3.
The availability of low cost embedded devices with TCP/IP and a Java Virtual Machine opens a new world to the previously mentioned closed network

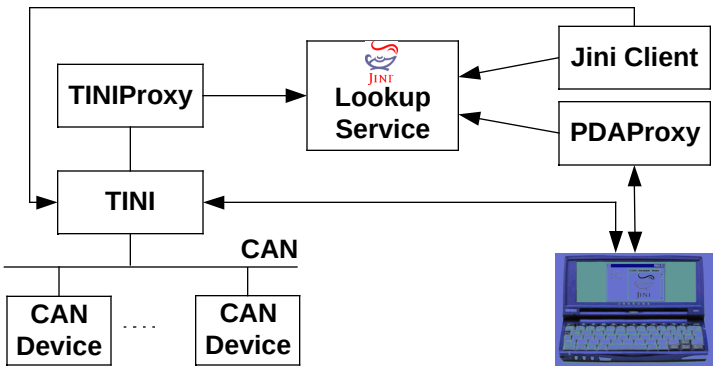

Figure 1 : System Overview

new world is Jini, which is a distributed system “allowing co-operating devices, services, and applications to access each other seamlessly, to adapt to a changing environment, and to share code and configuration transparently” [4].
The most popular example of Jini is the following. If someone links a Jini-enabled laptop to the network of a Jini-enabled conference room the laptop will automatically find the services of the conference room, like for example a print service and the laptop will automatically download any required drivers and be able to use the print service. If you go into another office, the services of the conference room discard and the new office services are discovered.
The significant benefit for the user are the dynamic integration and use of Jini-enabled services resp. devices. The user doesn’t have to install any device specific software or driver, as the code, to use a specific device, is downloaded dynamically. How can devices on a Controller Area Network benefit from Jini ?
The general ideas of services in Jini and devices on CAN are quite similar. Devices on CAN are plugged together with the goal to fulfil some kind of control. These devices share and exchange information over the network. Spoken in terms of services, each device is a special kind of (hardware) service, which offers its services to other devices (and services) on the network. An I/O-module for example offers the service to set its output lines and to read its input lines. Each module has some configuration data and a well known interface (i.e. communication protocol). As far as the application layer protocol CANopen is concerned, the interface comprises among other things service data objects (SDO) and process data objects (PDO) to exchange information.
The potential impact of Jini is compelling and as stated in [16] “Jini is here to give an object-oriented interface to the computer of the future; federations and services connected by a network”.
In the following section we give a short introduction to the Jini technology. Section 3 gives an overview of the system’s hard- and software architecture. Finally, an outlook on the future potential of Jini is given.

2. Jini Technology

The most important concept of Jini are distributed services. A service can be either a piece of software, hardware or a combination of both. A collection of services on a network is called ***Jini community or djinn***. The centre of a Jini community is the ***Jini Lookup Service*** [8]. The lookup service itself is a Jini service and serves as a repository of services. ***The Jini Lookup Service Browser***, depicted in figure 2, is part of the Jini implementation from Sun Microsystems and can be used to browse and manage currently registered services. Figure 2 shows the service ***ServiceRegistrar***, which represents the Jini lookup service, along with the corresponding service item.

Figure 2: Jini Lookup Service Browser

The ***Jini discovery and join protocols*** [7] enables services to locate and to join a lookup service. During the join process ***a service object*** (or ***service proxy***), and optionally some attributes, are stored by the lookup service. An entity which wants to use a particular service first searches for a lookup service and requests all services which map a particular type. If such a service exists, the service object is downloaded to the requesting entity. “This

devices without doing any explicit driver or software installation" [13].
Because "the ability to dynamically download and run code" is part of "the Jini architecture" it is assumed "that each Jini technology-enabled device has some memory and processing power" [6].
This includes that devices which want to participate in a Jini community have some processing power to run a Java Virtual Machine (JVM) and memory to store the Java classes. The ***Jini Device Architecture Specification*** [9] describes three different approaches for implementing a Jini service in hardware. The first one describes devices with resident JVMs, which are capable of running a full JVM including any resource necessary to participate in a Jini community. This approach results in higher costs, because of the existence of a microprocessor running the JVM and some memory to store the Java classes.
The second approach is concerned with devices using specialized Virtual Machines. The main idea is to implement only interfaces to the discovery and lookup services and some other functionality. This specialized JVM would result in limited functionality, but would be outweighed by the simplicity of such a device.
The last approach allows multiple devices to share a full JVM which serves as a proxy to the Jini community. With this approach, a group of devices is connected to an additional device either physically or over the network. Such a device is called ***Jini device bay*** which provides power, a network connection, and a processor running a JVM. As part of the registration of a new device at the device bay, the new device would tell the bay where to find the Java code needed to use the device. Upon registration at a Jini community, this code would be uploaded to the lookup service. The protocol used between the bay and the devices is not specified. With this approach, the devices themselves don't need additional hardware (CPU, memory), but there must exist an additional hardware device which manages the additional proxy functionality.
Sun Microsystems' implementation of Jini requires the Java 2 Standard edition, because of the Remote Method Invocation (RMI) activation framework and some classes which are not available prior to the Java 2 platform.

3. System Architecture Overview

The presented system is similar to the last approach from section 2, as a proxy for a group of devices connected to the CAN interface of an embedded system is used. Additionally, some effort had to be taken to tie up the embedded system with the network proxy. This results in an additional level of indirection between the Jini proxy and the CAN devices. It would be desirable to directly interconnect the embedded device and Jini, but at the moment the JVM of the embedded system doesn't offer this functionality.

Hardware Architecture

The hardware consists of an embedded system called ***TINI*** [10], a personal computer, used as a proxy to the Jini community, and optionally a personal digital assistant (PDA) running the client.
The TINI platform is a low cost embedded system from Dallas Semiconductor running a small JVM. The hardware consists of a 16-bit microprocessor, 512 KB of memory, an Ethernet controller and some peripherals. These peripherals include serial, parallel, 1-Wire ports (I2C), and two onboard CAN controllers.
The software on TINI includes a small operating system, a JVM, and the Java classes. The operating system provides a file system, memory management, I/O managers and task scheduling. All tasks, except the garbage collector are Java applications. The JVM on TINI is based on Java 1.1 and provides an additional package for I/O capabilities (e.g. CAN). Another interesting point is that "the goal of the TINI platform is to fully support Jini technology" [11] and the Java 2 Specification, including remote method invocation (RMI). For the moment, we have to cope with the restricted capabilities of the TINI JVM, which doesn't support RMI. This leads us to the current software architecture of integrating devices into the Jini community, simplified shown by figure 3.

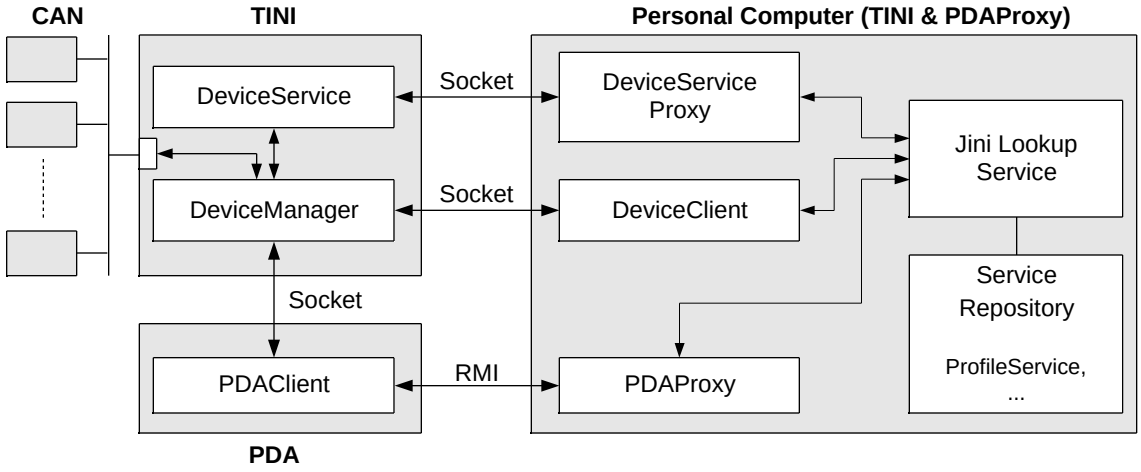

Figure 3: System Software Architecture

Software Architecture

In the following section, we use the terms DeviceManager-/Service/Proxy instead of CanOpenDeviceManager-/Service/Proxy, as they are just implementations in the more general framework.
The main components of the infrastructure include the DeviceManager and the DeviceService on the TINI system, and the DeviceServiceProxy residing on a personal computer running a full JVM including the packages needed to join the Jini community. Regarding the device architecture specification, a device proxy for the TINI system was implemented, which handles the Jini tasks on behalf of TINI and its connected devices.
The communication between the DeviceService on TINI and the DeviceServiceProxy is implemented using socket communication. This is similar to a traditional client-server system with the DeviceService as the client and the DeviceServiceProxy as the server.

TINI Software Components

The software on the embedded system primarily consists of two components, the DeviceManager and the DeviceService. The first one has the responsibility to manage the devices which are connected to CAN. To discover any devices on CAN, an application layer protocol is recommended. As a matter of principle any higher level protocol which supports

used. To accomplish the communication with CANopen modules, a subset of the CANopen protocol was implemented. Currently the system supports service data objects (SDOs) and some network management services. As TINI already supports CAN communication on the Java level, there is no need to interface any system driver with Java like described in [5]. The CanOpen-DeviceManager is an implementation of the interface DeviceManager, holding a list of the currently connected CANopen devices. To accomplish this task, the manager periodically sends SDOs with different identifiers and the standardized mandatory profile entry ‘devicetype’ to CAN. If the manager gets a response, it sends an additional request for the name of the device. Supposing a device with a node identifier of 8 exists, the manager first stores the identifier and type of this device in the device list. If the second request is successful, it also stores the name of the device. When a device discards, the manager observes during the periodic discovery process that a primarily detected device doesn’t exist any longer and deletes the according entry.
Until now, the work of the manager affects only local resources. To propagate the process of discovery and discarding of devices, the DeviceManager throws an appropriate event to interested listeners. This is implemented according to the ***observer pattern*** described in [12]. The DeviceService implements the listener

propagates the event by sending a specific message to the DeviceServiceProxy. The message includes the kind of event (DeviceDiscovered, DeviceDiscarded, etc.), the identifier, the name, and some other information. As mentioned before, the message is sent by sockets, as RMI is not available.

Network Proxy Components

The DeviceServiceProxy, residing on a personal computer, represents the network proxy of the DeviceService. This DeviceServiceProxy manages, like its counterpart on the embedded system, a list of the currently registered devices.
In the case of discovery, the DeviceServiceProxy creates a new object called DeviceProxy which implements the interface Device and registers this object along with some attributes with the Jini lookup service as described in section 2. After successful registration, the DeviceServiceProxy receives a universally unique identifier (***UUID***) from the lookup service, which is further used to identify the device in the Jini community. Figure 4 shows the previously mentioned lookup service browser with two registered devices along with the service item for one device. The three attributes ProfilePanel, DocumentPanel, and DevicePanel, shown in figure 4, are graphical user interfaces

Figure 4: Lookup Service Browser with two

stored along with the service object and will be described later.
If a device discards, the DeviceService-Proxy first examines if the device is registered as the state on the proxy system can sometimes differ from the real state. This kind of inconsistency can occur on system bootup when the registration process is in progress or if the devices are connected and disconnected within a short period of time.
Apart from the DeviceServiceProxy other services like the ***ProfileFileService*** and the ***ProfileService*** are registered in the Jini community. The ProfileFileService is responsible for delivering profiles (e.g. electronic data sheets) which are either stored in the file- or in a database system. The format of the profiles is not specified and can be exchanged with any other representation, like for example XML, as described in [14].
A typical client of the ProfileFileService is the ProfileService which knows how to manage (parse) a certain profile. In the case of an XML representation, there is the need for a ProfileService which can manage XML profiles. A client of the ProfileService receives a preprocessed data structure, which can easily be integrated into an application.

Client Components

The clients have to be distinguished between clients running a full JVM, like for example personal computers or laptops and clients running a restricted JVM like personal digital assistants. The latter lacks, among other things, of the ability to directly join the Jini community. To access the system infrastructure, two different clients are to be implemented. These are a PDADeviceClient (PDAClient) which is able to run on a restricted JVM, and a DeviceClient which is dependent on a Java 2 compliant JVM.
In the prototype implementation, a PDA with a PersonalJava Implementation from Sun Microsystems is used. The ***PersonalJava Application Environment Specification*** [15] defines a subset of standard Java and specifies mandatory and optional packages. The optional package RMI is part of the Personal Java

procedure to tie up the system to Jini is nearly the same as before. A network proxy, called ***PDAProxy*** running on a personal computer with a full JVM handles the Jini tasks on behalf of the PDAClient. This time we don't have to care about communication details as we are able to use the features of RMI. The PDAProxy connects to the Jini community through the discovery protocol. After it has discovered a lookup service, it requests all services which implement the interface ***Device***. If such a service exists, the appropriate service proxy is downloaded to the PDAProxy. As any device, which matches a particular type, is of interest, the PDAProxy registers itself at the lookup server to receive notifications about service registrations of the given type. Whenever such a service is registered or unregistered, the PDAproxy receives a notification. Equivalent to the services mentioned before, the PDAProxy stores a list of the currently available devices.

The first step taken by the PDAClient to participate in the system infrastructure, is to search the PDAProxy which is in fact an RMI server. This lookup is managed by the RMI registry, the naming service of RMI. In contrast to the naming service of Jini, the location of the RMI registry has to be known. After this lookup, the PDAClient is able to directly communicate with the PDAProxy through remote method invocation. The client connecting the PDAProxy requests a list of the currently registered devices and dynamically downloads the code to interact with these devices. This code includes a communication interface, necessary to directly communicate with the DeviceManager located on the embedded system. The communication interface is based on socket communication and handles all the communication between the client and the DeviceManager. After connecting the DeviceManager, the client has the ability to access the chosen device. The graphical user interface (GUI) of the PDAClient uses the abstract windowing toolkit (AWT) which is also part of the PersonalJava implementation.

The (Java 2) DeviceClient directly joins the Jini community and handles all the proxy functionality on its own. The process to events, and to download the service proxy is the same as described above, integrated in one application. In addition, the Java 2 client has a more sophisticated user interface as it can take full advantage of Swing. The Java Swing packages provide a powerful set of GUI components with a pluggable look and feel.

Furthermore, the DeviceClient makes use of an additional feature of Jini, the ability to store service attributes (entries) along with the service proxy. These attributes can contain additional information about the service, like for example a name, a location, as well as any other serializable object. Serializable means, that an object is turned into a sequence of bytes, which can later be restored fully into the original object. Therefore, it is also possible to store serializable user interfaces along with service objects. These objects have to be serializable, as they are downloaded by clients. The Java 2 client makes use of this feature, as it loads the appropriate user interface along with the service object.

As mentioned before, three user interfaces (implementing javax.swing.JPanel) are stored along with the service proxy. Figure 5 shows the profile tree (ProfilePanel) for a CANopen device.

Figure 5: Device Client (EDS-Profile)

The other interfaces (DocumentPanel and DevicePanel) are used to access a device (e.g. read/write SDOs), and to show a device specific document in HTML format.

4. Summary and Outlook

As the serviceability and manageability of future systems gets more and more

interfaced to some kind of service infrastructure. The Jini technology offers an appropriate framework to integrate devices and legacy systems into a dynamic infrastructure. As nowadays no systems, resp. devices supporting CAN and Jini are available, a low cost embedded system with CAN was integrated into the general Jini framework using proxies. Systems directly supporting Jini will be available in the near future. If we may think of the availability of CAN devices with some memory and processing power connected over realtime Ethernet, it would even be possible for these devices to directly participate in a Jini community. With the possibility to download code dynamically, no drivers have to be installed and all device specific interfaces (e.g. user interface) could be downloaded from the devices. The strict separation of the specification and the implementation of services would offer a huge potential for the creation of open standards regarding the serviceability and manageability of devices.

References

[1] G. Gruhler, G. Nusser, D. Bühler: Teleservice of CAN Systems via the Internet, Proceedings of the 6th International CAN Conference (ICC '99), November 1999, Torino, Italy. CAN in Automation (CiA).

[2] R.-S. Schimkat, G. Nusser and D. Bühler: Scalability and Interoperability in Service-Centric Architectures for the Web. In 11th International Workshop on Database Experts Systems Applications (DEXA 2000), September 2000, London-Greenwich, UK. IEEE Computer Society Press. (to appear)

[3] D. Bühler, G. Nusser, G. Gruhler, W. Küchlin: A Java Client/Server System for accessing Arbitrary CANopen Fieldbus Devices via the Internet, South African Computer Journal, No. 24, Nov. 1999, p. 239-243, ISSN: 1015-7999.

[4] S. Oaks, H. Wong: Jini in a Nutshell, O'Reilly & Associates, March 2000.

[5] D. Bühler, G. Nusser: The Java CAN API – A Java Gateway to Fieldbus Communication, Proceedings of the 3rd IEEE Workshop on Factory Communication Systems (WFCS 2000), Sep. 2000, Porto, Portugal. IEEE Computer Society Press. (to appear)

[6] Sun Microsystems Inc.: Jini Architecture Specification, Revision 1.0, January 1999.

[7] Sun Microsystems Inc.: Jini Discovery and Join Specification, Revision 1.0, January 1999.

[8] Sun Microsystems Inc.: Jini Lookup Service Specification, Revision 1.0, January 1999.

[9] Sun Microsystems Inc.: Jini Device Architecture Specification, Revision 1.0, January 1999.

[10] TINI Homepage. http://www.iButton.com/TINI

[11] R. D. Giorgio, D. Loomis, S.M. Curry: A promise of easier embedded-system networking. November 1999. http://www.javaworld.com

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

[13] W. Keith Edwards: Core Jini, Sun Microsystems Press, 1999.

[14] D. Bühler, G. Gruhler: XML-based Representation and Monitoring of CAN devices, Proceedings of the 7th International CAN Conference (ICC 2000), Amsterdam, Netherlands, October 2000. CAN in Automation (CiA). (to appear)

[15] Sun Microsystems Inc.: PersonalJava Application Environment Specification, Version 1.1.2, August 1999.

[16] Bill Venners: The Jini vision. August 1999. http://www.javaworld.com

Acknowledgements

This work is partially based upon work within the research consortium VVL funded by the Ministerium für Wissenschaft, Forschung und Kunst of the state of Baden-Württemberg through the research initiative Virtuelle Hochschule.

---

[1] University of Tübingen
WSI – Symbolic Computation Group
Sand 13
72076 Tübingen, Germany
nusser@informatik.uni-tuebingen.de

[2,3] University of Applied Sciences
Reutlingen / STA Reutlingen
Alteburgstr. 150
72762 Reutlingen, Germany
Phone: 0049 7121 271331
Fax: 0049 7121 25713
gerhard.gruhler@fh-reutlingen.de
http://www-sta.fh-reutlingen.de
http://robo16.fh-reutlingen.de