Teleservice of CAN Systems via Internet

Gerhard Gruhler [1,3], Gerd Nusser [1,2], Dieter Bühler [1,2], Wolfgang Küchlin [2]

[1] Institut für angewandte Forschung in der Automatisierung (IFA), Fachhochschule Reutlingen
[2] Wilhelm-Schickard-Institut für Informatik (WSI), Universität Tübingen
[3] Steinbeis-Transferzentrum Automatisierung (STA), Reutlingen

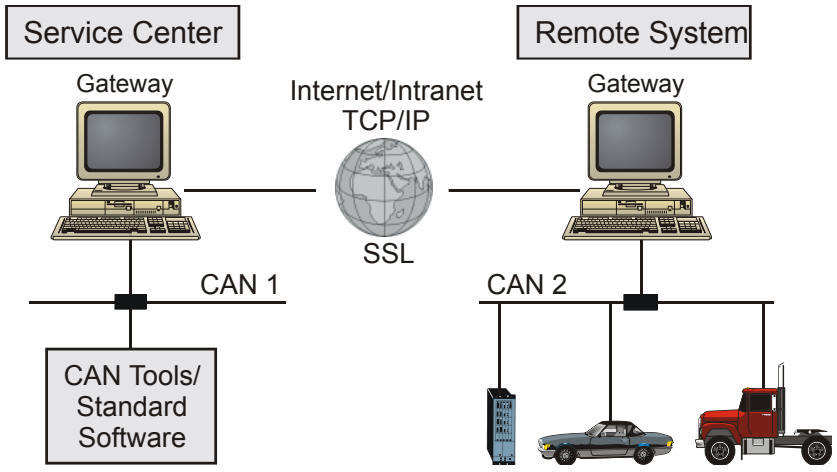
This paper introduces a system which offers the possibility to access remote CAN devices over the Internet as if they were local. The system consists of at least two identical gateways connected over the Internet. Each of these gateways is connected to a CAN bus with at least one CAN device. This architecture offers the opportunity to use locally installed standard CAN tools to maintain remote CAN devices, independent from any higher layer CAN protocol which is used. There is a huge application potential not only for automation components but also for diagnostic purposes in cars and trucks.
In order to ensure secure information transfer over the Internet, the system encrypts the transferred data on demand by using the secure socket layer protocol. Because most of the system is written in Java, it is possible to reuse nearly the whole system on different platforms.

1. Introduction

Today, fieldbus systems are in use not only in industrial automation but also in service equipment, in health care and even in a lot of end-user products. The spectrum reaches from production lines over building automation to cars and trucks and many other applications. Worldwide access to these objects (resp. their built-in devices) independent from their geographical position would offer a huge potential for remote control, remote diagnostics and maintenance and even remote teaching[1]. The World Wide Web (Internet) is, as the name applies, an appropriate transfer medium because of its world-wide extension at low costs. Fieldbus systems may be used either in static equipment (e.g. production line) or in mobile applications (e.g. car). The technical progress makes it possible to access the Internet from (nearly) every geographical position either by wire or wire-less.

As [1] points out, "remote monitoring, remote control, remote maintenance, servicing and customer support, are very important fields in automation industry today". There is the need to access process information over large distances. Furthermore it would be most attractive to access this information with

Figure 1. Teleservice System Overview

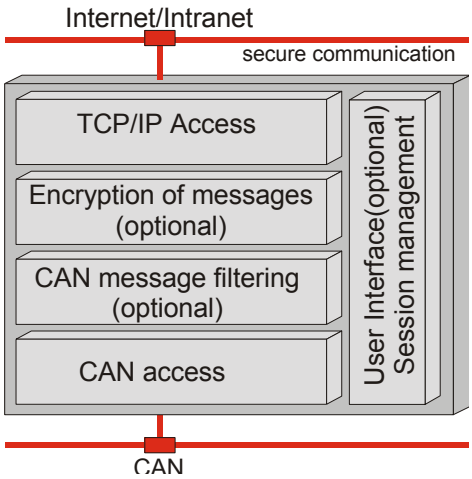


[1] The research project "Verbund Virtuelles Labor" which is funded by the Baden-Württemberg Research Ministry applies parts of the presented technology for remote teaching by access to physical devices.

Fachhochschule/STA Reutlingen
Alteburgstr. 150
Phone: 0049 7121 271331
Fax: 0049 7121 25713
gerhard.gruhler@fh-reutlingen.de
http://www.fh-reutlingen.de/~www-sta/

standard tools and standard hardware with no need of proprietary technologies. In fact "there is a gap between traditional fieldbus systems and Internet-based applications which has to be overcome" [2].
To close or at least minimize the gap between these network technologies there are three solutions. First it would be possible to equip fieldbus devices with an Ethernet interface, which would result in higher costs because of additional hardware (controller, wire, etc.) and software (TCP/IP stack) requirements. Secondly, replacing one technology with the other would only be practicable when the fieldbus technology would by replaced by Ethernet. As far as CAN is concerned, this would mean to ensure hard real time capabilities on Ethernet which can not be guaranteed without additional hardware and another network topology (token ring, switched Ethernet). At last, every technique is used in the field where it is meant to be used and the gap is closed by software resp. hardware.
Our remote access system uses CAN on part of the fieldbus and TCP/IP on part of the Internet. The main reason behind this decision is to use standard hardware and software components as mentioned in [3].
The hardware setup consists of two standard PCs each equipped with an Ethernet interface and a CAN interface. The existence of a second CAN bus is not compelling if the CAN driver offers so called virtual channels (i.e. data is not actually written to the hardware) which can not be assumed a priori.
To keep the system small, portable and open, we use free and standardized software whenever possible. Except for the system dependent parts (i.e. system driver software) the system is written in Java. Besides its platform independence, Java provides built-in network capabilities. Furthermore the system presented in this paper, can easily be used in a small embedded system. The requirements for the system consist only of an embedded Java virtual machine, an Ethernet and a CAN interface with appropriate drivers and protocol stacks (i.e. TCP/IP, CAN).
As the main purpose of the Internet is to transfer data from a specified source to a specified sink with a specified protocol in a user transparent manner, it is often depicted as a black box. It's hard to track down the route data takes and which institutions are responsible for specific subnets. We have to consider that the Internet is not secure without additional precautions. To accomplish identification and secure data transfer, the secure socket layer (SSL) protocol is used.
Figure 2 shows the major functions of the CAN/Internet gateway used in Teleservice Systems.

Figure 2. Major functions of a CAN/Internet gateway



2 System Architecture Overview

The hardware setup consists of two standard PCs (Intel Pentium, Windows NT) each connected to a CAN with at least one CAN device. The gateways are running as Java applications in order to have access to system specific hardware drivers.
The system is not limited to CAN or a specific application layer protocol because TCP/IP is only used as a transport mechanism. As described in the following section, CAN can be replaced by any other fieldbus offering a corresponding interface.

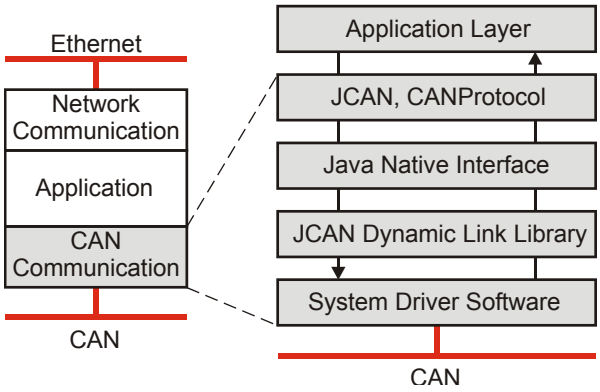
2.1 CAN Communication

The following sections describe the low level communication, shown in figure 3, in a bottom-up manner, starting with the developed system library (i.e. JCAN.DLL) used to communicate with the system-dependent driver software.

2.2 JCAN.DLL

The system-dependent driver software supports primitive access to CAN, providing simple open, close, transmit, receive and

Figure 3. CAN communication level



control (e.g. baudrate setting) functions.
The JCAN dynamic link library, as shown in table 1, handles the following two main tasks: first, it encapsulates the CAN driver, providing an abstract, thread safe view of the net traffic on CAN, on the other hand it is part of the implementation of the Java CAN API, which

Table 1. JCAN.DLL interface (extract)

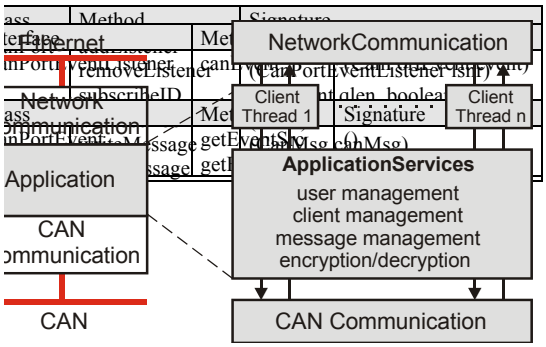
Function	Signature
initDriver	(long bitrate, int channel, long timer)
startMessageManager	()
stopMessageManager	()
transmit	(int id, int dlc, int flags, byte[] data)
receive	(int id, byte[] data, long[] timeStamp)
peek	(int id, byte[] data, long[] timeStamp)
subscribe	(int id, int qLength, boolean bNotify)
unsubscribe	(int id)

references the methods of the JCAN.DLL via the Java Native Interface. Therefore the JCAN dynamic link library offers a level of abstraction to the system-dependent driver software.
The main part of the JCAN dynamic library is the so-called message manager thread, which collects all CAN messages on the bus in an infinite loop. The CAN messages are managed with respect to their actual identifier. The library can be configured dynamically to hold specific message queues for each identifier. When a new message appears on the bus, the message manager retrieves the ID of the message and checks if some Java instance has subscribed (i.e. express some interest in receiving a certain message with the corresponding message identifier) for messages with this specific identifier. In this case, the message is stored in a queue (FIFO) reserved for messages with this identifier. In the other case the message is ignored.
A subscription for CAN messages takes three arguments: the identifier, the length of the message queue and the notification flag. The value for the queue length parameter depends on the characteristics of the messages. If the message history is important and no message must be missed, the queue length should be chosen high enough to hold all messages until they are read from outside the library, even if the receive frequency is comparably high. If it is not necessary to track all messages, a queue length of one may be appropriate, resulting in only the most recent message being available at any time. If the queue overflows, the oldest message is deleted.
The *initDriver* function initialises a message channel with a given transfer rate.
To read and write CAN messages from and to CAN, the *receive* (resp. peek) and *transmit* functions are used. The *receive* function reads the oldest available message from the queue specified by the identifier *id* and removes the corresponding message from the queue. The *peek* function has the same functionality as *transmit* without removing the message.
The message manager thread collects messages which were previously specified by *subscribe* and ignores further messages by *unsubscribe*.
The Java instance has the possibility to set the queue size for these messages and to choose between asynchronous notification and polling techniques to read incoming messages. Since unsubscribed messages are ignored, it is possible to mask the bus traffic which is not of interest.
The decision to implement this message management in C++ results in a high performance filtering on incoming CAN messages, keeping the Java side of the CAN communication package away from the time critical aspects of the CAN message management.

2.3 Java CAN API

The Java CAN API represents the Java part of the CAN communication level. It provides some classes and interfaces to transparently access the hardware driver and therefore offers another level of abstraction to the underlying hardware. This results in an independent way to access hardware from different vendors. Our concept is suitable for all typical field busses, but we will use CAN as our standard example.
The CanPort class, as shown in table 2, provides the native method declarations for the JCAN dynamic library and methods for the CAN event handling. To access a CAN bus, a Java instance simply needs a reference to a

Figure 3. Gateway as a concurrent server



CanPort object which is further used to communicate with CAN (resp. with the JCAN proxy and the corresponding message manager thread).

Table 4. CAN protocol package (extract)

Class	Method	Signature
CanMsg	setId	(int id)
	getId	()
	setData	(byte[] data)
	getData	(byte[] data)

Java instance need to implement the *CanPortEventListener* interface (table 3) by overriding the *canEvent* method and sign itself as an event listener at the *CanPort* object using the *addEventListener* method.

This technique is implemented according to the observer pattern as described in [4] which is a standard feature of Java.
If the message manager encounters a subscribed message with the notification flag set, all registered CAN event listeners are notified by invoking their version of the *canEvent* method. The *CanPortEvent* argument passed to the *canEvent* method contains information about the event type and the event source so the registered Java instances can react in an appropriate way, e.g. call *receive* or *peek* to read a message from the queue. The message format itself is encapsulated in the CanProtocol package. The CanProtocol package, as shown in table 4, comprises classes for generic CAN layer 2 messages and CANopen layer 7 messages.

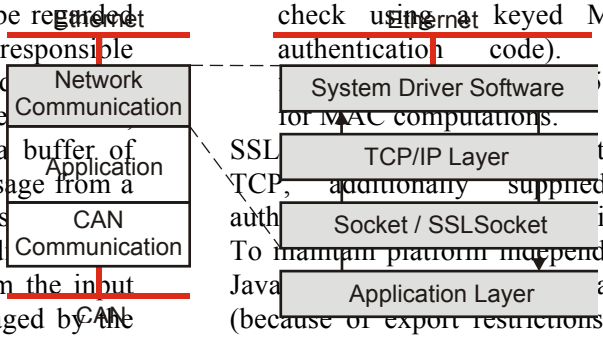
The latter will not be discussed within the scope of this paper. The *CanMsg* class encapsulates a generic CAN message with a 11 bit (CAN 2.0A9 or 29 bit (CAN 2.0B) identifier, 8 bytes of data and the data length code (dlc). A *CanMsg* is read resp. written by the corresponding methods of *CanPort*.
In general, the classes provided by the CanPort and CanProtocol packages in conjunction with the JCAN.DLL offer a convenient way for CAN communication.

2.4 Network Communication

The Teleservice System is designed as a client/server architecture [5] using Java sockets (class *Socket*, class *ServerSocket*) which are based on TCP [6]. TCP provides a connection-oriented, reliable, full duplex, byte stream service [7]. Because sockets are based on the transport layer of the ISO/OSI reference model [8], they represent the lowest level of network communication accessible from an application program. The abandonment of a middleware software layer, for example DCOM/OPC technology ([9],[10]), the Common Object Request Broker Architecture (CORBA [11]) or Java RMI ([12]), benefits in very few system requirements and few communication overhead. However, it is possible to equip the system with some kind of middleware technology. This was already part of former developments and is extensively presented in [2].
To handle multiple requests at the same time, the server was realized as a concurrent server as depicted in figure 4. Each client request is managed by a single Java thread. The problem arising, when multiple threads want to access a single CAN bus will be treated in section 3.

Besides the actual CAN message transfer, the communication protocol between a client and a server contains messages for user management (e.g. login procedure), application management (e.g. start message transfer), and message management (e.g. message subscription).

Figure 5. Network communication overview



As mentioned previously, the communication between a client and a server can be regarded as a stream of bytes. The classes responsible for input (*InputStream*) and (*OutputStream*), resp. any derive offer methods to read and write a buffer of bytes to sockets. To transfer a message from a source to a sink, a buffer of bytes is the output stream of the correspond At the sink, the buffer is read from the input stream. The further action is managed by the application logic.

Because neither sockets, nor the streams used for input and output, nor any of the underlying layers modify the transferred buffer, it is transferred in the same form as it is written. Thus, anybody who listens to the data, has the ability to interpret the messages and the protocol used between client and server.

To ensure privacy, we have to take precautions, which are discussed in the following section.

2.5 Secure Network Communication

The great flexibility of TCP/IP has led to its world-wide acceptance as the basic Internet and Intranet communication protocol. According to [13], information on the net can be interfered in the following ways:

- Eavesdropping. Information remains intact, but its privacy is compromised.
- Tampering. Information in transit is changed or replaced.
- Impersonation. Information passes to a person who poses as the intended recipient.

The leading protocol for providing a secure data transfer over the Internet/Intranet has been developed by Netscape Communication Cooperation and is called Secure Sockets Layer (SSL). The SSL protocol, now standardized as Transport Layer Security (TLS [14]), supports peer authentication, data encryption and data integrity. The connection security provided by SSL has three basic properties ([15]):

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g. DES, RC4, etc.).
- The peer's identity can be authenticated using asymmetric, or public key cryptography (e.g. RSA, DSS, etc.).
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC (message authentication code). Secure hash , etc.) are used for MAC computations.

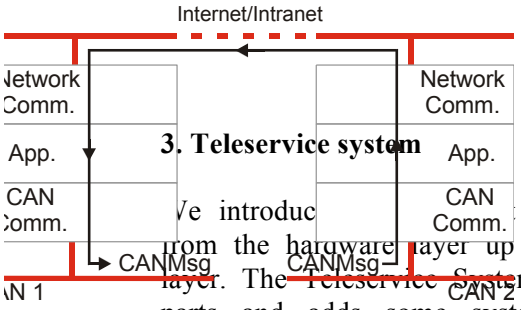
SSL terface used for TCP, additionally supplied with peer auth ion.

To maintain platform independence, we use a Java an vendor [16] (because of export restrictions) based on the "Java Cryptography Architecture" (JCA [17]) provided by Sun Microsystems, which offers a framework (a set of interfaces) to access and develop cryptographic algorithms. An implementation of the JCA is called "Java Cryptography Extension" (JCE [18]). The package provides an implementation of the JCA and an implementation of SSL sockets, which can be used like standard Java sockets except for additional initialisation steps. As far as the communication between two systems is concerned, the code stays the same. As before, the data is transferred using input and output streams. The major difference with use of SSL sockets is, that the data which is written to an output stream is encrypted and later decrypted by the input stream. The different layers of the network communication level are depicted in figure 5.

Even though in the current system the type of communication (standard or secure) must be given at start up, it would also be possible to change the communication behaviour dynamically by the use of two communication channels with different strategies.

To sum it up, the SSL protocol offers a high degree of security ([19]) paired with little effort and acceptable overhead ([20]).

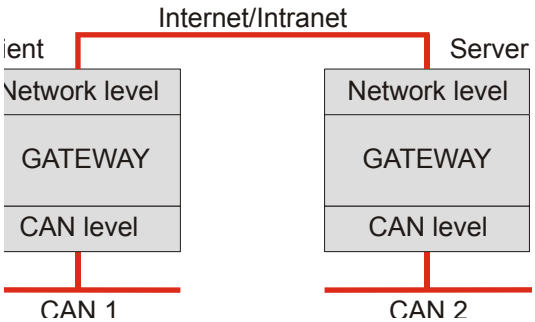
Figure 7: Transfer of CAN Messages



3. Teleservice system

We introduce gateway layers from the hardware layer up to the network layer. The Teleservice System comprises all parts and adds some system (e.g. client synchronisation) and user management facilities. The Teleservice system consists of two identical Java applications as depicted in figure 6.

Figure 6. Teleservice System



The behaviour of these applications differ in so far as one application takes the role of a client and the other takes the role of the server. The corresponding behaviour has to be specified at startup.

It is often not obvious which application plays which role. We distinguish the following different situations. Imagine a situation where for example a service center waits until it is connected by clients because of a problem. For example there is a problem in a factory and the responsable machine informs the service center. In this case the service centre plays the role of a server. Another case could occur when a person wants to observe some remote machines and therefore plays the role of a client. The difference between these situations is that in the first case the machine is connected to the client and in the second case it is connected to the server.

When a client connects to a server, it has first to authenticate itself by username and password. The supplied information is used to determine system access and the corresponding access rights (read, write, read/write). If the authentication procedure was successful, the client grants access to the system, otherwise it will be rejected.

The login procedure is followed by a subscription procedure where the client can subscribe either to all or only to a subset of CAN messages which will be observed. The specified set is then transferred to the remote system and passed to the message manager thread as described in section 2.

After this initialization the following steps are performed:

1. The message manager thread on the remote system receives a message according to the subscription mask and passes it up.
2. The application takes the message, builds a packet (buffer) and sends it over the network.
3. The client receives the packet, extracts the message and writes it to CAN.

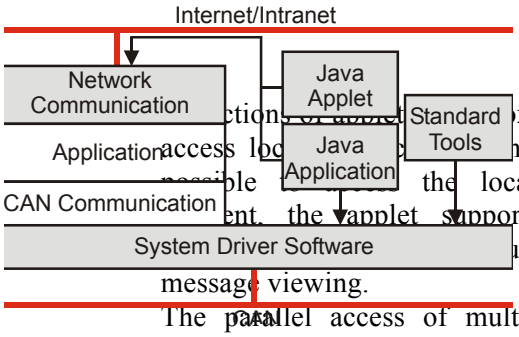
Writing messages to a remote bus is treated in the same way. As far as data transfer is concerned, it can be chosen between secure and standard communication at system start up. The transfer of CAN messages from one gateway to another is depicted in figure 7.

From the point of view of a standard application, the whole process of transferring messages from a remote machine to the local machine is completely transparent. Such an application observes only the local system driver which is supplied by remote messages. Thus the local application takes these messages from the driver and acts with respect to its strategy (e.g. message logging). When an application writes messages to the driver, the messages are transferred to the remote system and are written to the remote system driver software.

In addition, a Java applet can be used to access devices on a remote system (user interface of figure 2). This applet can be used in any browser which supports Java (e.g. Netscape Communicator, Internet Explorer). To use applets, an additional webserver has to be installed on the system which is connected to the fieldbus.

The scenario of the initialisation process is the same as mentioned before. Because of security

Figure 8: Access to the Teleservice System



...x model), it can't access loc... herefore it is not ...ible ... the local bus. At the ...ent, the applet supports remote user ...bscription and message viewing.

The parallel access of multiple client to a single system can be devided into user management and message management. As far as user management is concerned, at most one client has write access but multiple clients have read access. The message management is treated by the JCAN dynamic library which simply serialises the incoming requests.

On the basis of the implemented software architecture it is little effort to implement further applets and applications. In general, it is possible to implement applets, which have the same functionality as the common standard tools to access remote devices. As an example, we have already implemented an applet which makes it possible to communicate with a remote CAN bus using CANopen. This applet, called "Java Remote CAN Control" supports access to CANopen devices, the corresponding device profiles, network management services (operational, preoperational) and writing resp. reading of SDOs. The system is available for public access via [21].

The various configurations to access the Teleservice System are depicted in figure 8.

The possibility to access either local or remote CAN busses in a transparent manner offers a great potential for further developments. However, it must be considered that real time behaviour is lost at the expense of the flexibility and distribution of the system. This disadvantage can be compensated if the system has the ability to log messages in a specified time interval. Upon expiration of this interval, the messages may be transferred and analysed. Perhaps it would even be possible to "playback" the messages in real time and send them to a local existent original assembly.

4. Summary and Outlook

In this paper we introduced a system which offers opportunity to access remote CAN busses as if they were local.

To close the gap between Internet and CAN, the JCAN.DLL and the Java CAN API developed for common access to CAN field busses form a general basis for any Java applications, hiding bus specific message layout and protocol issues. High performance message filtering and asynchronous notification is supported to create a convenient interface to the CAN bus traffic. Since only standard libraries and packages were used, the effort to adjust this system to different CAN driver APIs or even different fieldbus drivers is very small.

The main advantage of the client/server communication using standard sockets is high availability with low system requirements and less overhead for communication.

To encrypt and decrypt data over the Internet, the SSL protocol was used. With this kind of protocol it is possible to establish secure connections, ascertain authentication and ensure originality of data.

Because most of the system, except the interface to the system-dependent driver software, is written in Java, it can be ported to any operating system with a Java 1.1 compliant virtual machine with little effort. The requirements could even be met by a low cost embedded system, featuring a small Java Virtual Machine equipped with CAN, an Ethernet interface and appropriate protocol stacks. The application spectrum reaches from remote teaching over industrial machines to buildings or even cars and trucks. For example, a truck driver experiencing a breakdown in the middle of nowhere, can use such a system linked to a mobile phone to establish a connection to the Internet to enable remote access to the CAN devices built into the truck, allowing a distant engineer to check the vehicle for malfunctions.

References

[1] G. Gruhler, W. Küchlin, and Th. Lumpp: „Accessing CAN-based automation systems via the Internet“, CAN Newsletter, vol.1, pp. 24-28,

March 1998

[2] Th. Lumpp, G. Gruhler, W. Küchlin: “Virtual Java Devices. Integration of Fieldbus Based Systems in the Internet”, Proc. of the 24th Annual Conference of the IEEE Industrial Electronics Society, IECON ’98, Aug.-Sept. 1998

[3] G. Gruhler, Th. Lumpp, W. Küchlin: “Zugriffskonzepte auf CANopen-Geräte über das Internet/Intranet”, VDI Berichte 1410, VDI/VDE-GMA und CiA Fachtagung CAN in der Automatisierung, pp. 77-89, 1998

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

[5] Andrew S. Tanenbaum: "Modern Operating Systems", Prentice-Hall Inc., 1992

[6] J. Postel: "Transmission Control Protocol", RFC 793, September 1981

[7] W. Richard Steven: "UNIX Network Programming", Prentice-Hall, Inc., 1990

[8] H. Zimmermann: “OSI Reference Model – The ISO Model for Architecture for Open Systems Interconnection”, IEEE Transactions on Communications COM-28, No.4, April 1980

[9] Microsoft Corporation, DCOM Technical Overview – White Paper, 1996

[10] OPC Task Force, OPC Overview. OPC Foundation, Version1.0, October 1998 http://www.opcfoundation.org

[11] Object Management Group (OMG), The Common Object Request Broker: Architecture and Specification, Rev. 2.1, Aug. 1997

[12] Sun Microsystems, Java Remote Method Invocation (RMI) http://java.sun.com

[13] Netscape Communications Corporation: "Introduction to Public-Key Cryptography", 1998 http://developer.netscape.com:80/docs/manuals/security/pkin/contents.htm

[14] T. Dierks, C. Allen: "TLS Protocol Version 1.0", RFC 2246, January 1999

[15] Netscape Communications Corporation, The SSL Protocol Version 3.0, Internet draft. http://home.netscape.com/eng/ssl3/ssl-toc.html

[16] Institute for Applied Information Processing and Communications Graz University of Technology (IAIK). http://www.iaik.tu-graz.ac.at/

[17] Sun Microsystems: "Java Cryptography Architecture" http://java.sun.com/products/jdk/1.2/-docs/guide/security/CryptoSpec.html

[18] Sun Microsystems, Java Cryptography Extension (JCE) http://java.sun.com/products/jce/

[19] Bruce Schneier: "Applied Cryptography: Protocols, Algorithms, and Source Code in C", John Wiley & Sons, Inc., 1996

[20] IAIK, “How fast is IAIK-JCE ?”, http://jcewww.iaik.tu-graz.ac.at/jce/howfast.htm

[21] "Java Remote CAN Control". Online Internet demonstration of accessing arbitrary CANopen devices. http://robo16.fh-reutlingen.de/doku/Demo/InterCanDemo.html