ejIP: A TCP/IP Stack for Embedded Java

Martin Schoeberl

Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Abstract

To enable Java on resource constraint embedded devices, the whole system should be implemented in a single programming language to avoid overheads on language boundaries. However, most of the functionality that is dedicated to the operating system layer is usually written in C. In this paper we present the design and implementation of a network stack written entirely in Java. This implementation serves as an example how to implement system functions in a safe language and gives evidence that Java can be used for operating system related functionality. The described TCP/IP stack ejIP has already been successfully used in industrial projects on pure Java embedded systems.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Network communication

General Terms Network Stack, Real-Time Systems

Keywords Java, Embedded Java, Real-Time Java

1. Introduction

Network connection is an integral part of many embedded systems. In traditional implementations the network code is part of the operating system (OS) and usually runs in kernel mode. In small embedded systems, with applications implemented in Java, it can be resource saving to omit the OS and run the Java virtual machine (JVM) on the bare metal [22]. The JVM becomes the OS. The JVM already provides some services (e.g., multithreading support) that are associated with an OS. Furthermore, as Java is a safe language with runtime checks, the expensive memory protection, which is part of the OS, can be avoided. With Java, type safety can be used instead of hardware for memory protection, resulting in a more lightweight system.

When we allow access to I/O devices from Java, e.g., via hardware objects [19], we can avoid crossing the language boundary between Java and C in the application. Only the bare JVM itself needs to be implemented in C or in hardware. Using a single language for the whole firmware and application stack simplifies development and testing.

To enable these *Java only* embedded systems we need to provide implementations of standard operating system libraries and drivers in Java. In this paper we describe the contribution of a Java based TCP/IP stack for such a system. The TCP/IP protocols and the drivers are all written in plain Java.

The classes of ejIP are organized around the TCP/IP protocol layers. The whole TCP/IP stack and the application logic can run in a single thread or can be split over several threads. The later organization is beneficial when executing the application on a chip-multiprocessor system. ejIP performs its own memory management of packet buffers and avoids generation of garbage at runtime. Avoiding garbage collection simplifies reasoning on the real-time behavior of a program.

As embedded systems are often real-time systems, the architecture of ejIP is designed to avid blocking operations. Blocking read or write operations prohibit worst-case execution time (WCET) analysis of tasks [25], which is mandatory for hard real-time systems. On a time-predictable platform (e.g., the Java processor JOP [18]), the TCP/IP stack ejIP is WCET analyzable. Using a single language for the whole embedded application enables analysis of the WCET. Calling a C based TCP/IP implementation via JNI would defeat such an analysis. There are no WCET analysis tools available that support mixed language systems.

In this paper we show that networking, which is usually considered system code written in C, can be implemented in a safer language like Java. Furthermore, we present an API to the network stack that is a better fit for embedded real-time systems. By substituting the stream based, blocking functions by periodic polling for network activity, the network stack becomes time-predictable and analyzable.

Following TCP/IP protocols are implemented in ejIP: IP, ICMP, UDP, and TCP. On top of those protocols ejIP implements TFTP and contains examples for various TCP/IP applications (e.g., a tiny web server, a telnet server, a SMTP client,..). For the link layer ejIP provides implementations for SLIP, PPP, and an Ethernet driver (including ARP) for the CS8900 chip. ejIP is in use in several industrial applications.

The paper makes following contributions:

- We describe the design of a fully Java implementation of TCP/IP for embedded real-time Java.
- The TCP/IP stack demonstrates that OS level code can be implemented in Java.
- The open-source design is a contribution to the development of Java only embedded systems.

The paper is organized as follows: the next section describes related work on TCP/IP stacks for embedded systems. Section 3 describes the design goals and the resulting design of ejIP. The usage and application interface of ejIP differs from socket based TCP/IP stacks and is described in Section 4. The evaluation in Section 5 describes use cases of ejIP and Section 6 concludes the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.
Copyright © 2011 ACM 978-1-4503-0935-6. . . $10.00

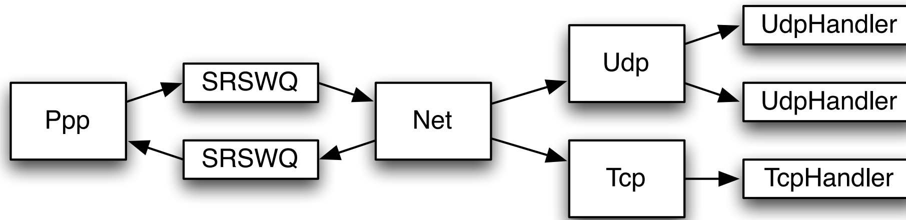

Figure 1. An example usage of ejIP with the link layer and upper layers in different threads. The example contains two UDP applications and one TCP application.

2. Related Work

Most standard TCP/IP stacks are based on the Berkeley TCP/IP implementation included in Berkeley's Unix [12]. This first Unix implementation that included an implementation of the Internet Protocol also introduced the BSD socket API for networking, which is now the de-facto standard API for networking and evolved into the POSIX socket API [10].

However, this TCP/IP implementation is written for desktop and server machines and usually to large for embedded systems. Therefore, several cleanroom implementations of TCP/IP for embedded systems have been developed. Furthermore, the BSD socket API needs multi-tasking support and therefore many TCP/IP stacks for small embedded devices provide a different API, which relies on polling and event driven applications. With ejIP we follow a similar approach for the API.

Probably the most successful TCP/IP stack implementations for embedded devices are *lwIP* (lightweight IP) and *uIP* (micro IP) [3]. The main difference between lwIP and uIP is the handling of TCP retransmissions. While lwIP keeps the packet buffers allocated for retransmission, applications using uIP have to reproduce the data on a retransmission. The later approach further saves memory. Both TCP/IP stacks implement an API that differs from the BSD sockets. The application contains a main control loop, which checks for arrived packets and timeouts. Both generate application events. A programming abstraction called protothreads [6] can simplify application programming for the event-driven uIP and lwIP stack. uIP has recently been extended to support IPv6 [7].

Microcontroller companies often provide a TCP/IP stack optimized for their product, e.g., Microchip's TCP/IP stack includes its own cooperative multitasking system [15]. Therefore, the user tasks have to be written in cooperative style and longer jobs have to be divided into multiple tasks. ejIP can be used in a similar fashion for a single-threaded runtime, but all operations in ejIP are correctly synchronized and multithreading is fully supported. An a chip-multiprocessor, such as the JOP CMP [14], different layers of the TCP/IP stack can run in multiple threads on multiple processors.

In order to reduce processing overhead in embedded systems, the Linux TCP/IP code has been adapted to avoid packet copying [2]. It has been reported that the processing overhead can be decreased by up to 63% and the object code size is reduced by 22%. To transfer data between user mode and kernel mode the virtual memory system needs to remap memory pages. With ejIP we also implement a zero copying TCP/IP stack, but due to the safety of Java no page remapping is needed.

Wireless sensor network (WSN) devices have extreme resource constraints. However, even for those small devices IP is now considered as preferred communication protocol [16]. Using IP in WSN is an important step towards the 'Internet of Things'. The uIP TCP/IP stack was one of the first implementations of TCP/IP for WSNs [4, 5]. Later implementations of TCP/IP for WSN adapt the TCP/IP congestion control algorithm to fit better for a network link with high packet losses [26].

The idea of using a safe language for OS related tasks is explored in Singularity [9]. Singularity is a research OS based on a runtime managed language (an extension of C#) to build a software platform with the main goal to be dependable. A small HAL (IoPorts, IoDma, IoIrq, and IoMemory) provides access to PC hardware. The Singularity OS uses device objects and interrupt handlers to enable device drivers in C#.

There are not (yet) so many *Java only* embedded systems available. Therefore, there are not that many Java implementations of TCP/IP available. One exception is the runtime library for the Java processor jamuth [24]. It contains a TCP/IP stack in Java that supports ARP, ICMP, UDP, and TCP.[1] As this TCP/IP stack is a closed source, commercial implementation, no implementation details are available.

The development of a Java only ecosystem for embedded devices is a chicken-egg problem. Without having all drivers available in Java source, a C-based OS is needed. Without a true Java only runtime there is no incentive to develop Java-based file systems, network stacks, and I/O device drivers. To help to solve this chicken-egg problem we provide the ejIP network stack in open-source. A Java port of the YAFFS (Yet Another Flash File System) file system for NAND Flash is also available in the JOP distribution.

3. Design

The first version of ejIP has been released around 2003, containing the basic protocols, but only a stripped down version of TCP to support a single HTTP request. Since then, ejIP has been extended with TCP support. The packet communication between the different layes has been reorganized to take advantage of chip-multiprocessors by using non-blocking queues. In the following section only the latest version of ejIP is described.

Figure 1 shows an example configuration of ejIP. The link layer for PPP communicates with the upper layers via two non-blocking queues (SRSWQ). Therefore, it can run in its own thread (or processor). Net dispatches received packets to Udp and Tcp. Those forward packets to the applications, which implement either the UdpHandler or the TcpHandler interface. The handlers are executed within the Net thread. If TCP/IP applications shall run on their own processor core, the non-blocking queues can be used to further delegate packet processing from the handlers to application threads.

[1] Private communication with Sascha Uhrig.

3.1 Design Goals

For a resource constraint embedded system we state following design goals for ejIP:

- Low memory footprint
- Compatible with real-time systems
- Avoid garbage collection

Low memory footprint is the one of the main constraints for embedded systems. The class files of the whole ejIP make up 104 KB. The largest classes are the link layer driver for PPP and Ethernet. An application usually includes only one of the link layer drivers. A compiled and linked application including ejIP and a simple HTML server needs 76 KB on JOP.

To use a TCP/IP stack in real-time systems two properties have to be fulfilled: the code needs to be WCET analyzable and the design needs to fit for the periodic task model of real-time systems. For the WCET analysis all loop bounds need to be known. For the periodic task model blocking on a resource shall be avoided.

Garbage collection (GC) is an essential part of the Java language. However, in real-time Java GC is usually avoided and the scoped memory model is used for temporal dynamic storage. In ejIP all buffers and data structures are allocated at the creation of the TCP/IP stack objects and no new objects are allocated at runtime. The buffers for the network data are reused and organized in a pool.

3.2 Safety-Critical Java

As Java Specification Request 302 (JSR 302) a new standard of Java for usage in safety-critical systems is proposed. Safety-Critical Java (SCJ) [8] is an extended subset of the Real-Time Specification for Java (RTSJ) [1]. SCJ has tow operation modes: mission initialization without real-time guarantees and mission execution where all tasks have to meet deadlines. The real-time tasks are organized as handlers that are releases periodically by the SCJ scheduler. Therefore, a SCJ application has to be organized as periodic tasks. The API of ejIP, with the polling network interface and handlers for the UDP and TCP packets, fits well into this model of computation.

SCJ disallows garbage collection. For allocation of temporal objects RTSJ style scoped memories can be used. However, those scoped memories are thread private and can not be used for communication between threads. All objects that are shared between threads have to be allocated at mission initialization time. The preallocation of the packet buffers at application start in ejIP fits well for SCJ. Packets are managed in a pool and no dynamic data is allocated at runtime.

3.3 The Components and Communication

The code is organized around the layers of TCP/IP. The main classes are:

Ejip represents one instance of the TCP/IP stack. An application can instantiate several independent TCP/IP stacks. At creation the maximum buffer sizes and the number of buffers can be set. The pool of free packet buffers is managed by Ejip.

Net creates the objects that implement IP, UDP, and TCP. Net implements a Runnable that has to be executed periodically. Within Nets run() method packets are received from the link layer and distributed to UDP or TCP. ICMP ping requests are directly handled within Net.

Ip is a passive class that contains utility functions for IP header generation and reading and writing data into the packet buffer.

Udp handles UDP packets. Applications using UDP have to register an UdpHandler with the receiving port number. When Udps process() method is invoked from Net, the packet is forwarded to the registered handler. If no handler is registered for the port, the packet is simply dropped. Udp implements a Runnable and within run() the handlers' periodic loop functions are executed.

Tcp handles, similar to Udp, the TCP packets. TCP applications have to implement the abstract class TcpHandler and register it with a port number. A TCP connection is represented by a TcpConnection. Tcp is also an active class and processes the TCP timeouts in the run() method. Tcp also invokes the handlers' periodic loop functions.

Tftp implements the Trivial File Transfer Protocol (TFTP). TFTP is a simple stop-and-go file transfer protocol on top of UDP. Therefore, the implementation is very lightweight. TFTP is primarily used to program the Flash to update the software in the field after deployment over the Internet.

At least one link layer object needs to be created. Each link layer class implements Runnable and run() has to be executed periodically to poll the receiving hardware. At the link layer following classes represent different protocols:

CS8900 represents a link interface to the Ethernet with the popular CS8900 chip. The CS8900 chip contains SRAM for packet buffering and is used in polling mode by CS8900. The code of CS8900 has been derived from the Linux driver for the CS8900 chip. For the Ethernet to Internet address translation the class Arp handles ARP requests and replies.

Slip implements the SLIP protocol and also handles the special variation of SLIP under Windows. Slip uses a class Serial for the access to the serial port. This class is device specific and has to be adapted to the target hardware and runtime system.

Ppp implements the point-to-point protocol (PPP). PPP was used in the early days of the Internet for modem access. Today mobile devices connected to the Internet via GSM or GPRS map the mobile network connection to PPP. This mapping is a little bit problematic. For example, user identification used by PPP is only simulated by the modem and connection errors (such as wrong password) are not detected at this stage, but later signaled by an unrelated error. Ppp also handles the modem interface with modem setup, dialing, and modem hangup.

Loopback is a simple loopback device. It moves packets from the send queue into the receive queue. The main usage of the loopback interface is for benchmarking of ejIP. ejIP is also part of JemBench [20], a benchmark for embedded systems.

The active objects exchange packets via non-blocking single reader/writer queues [11]. In the original version of ejIP the communication between the different components was handled via a packet pool. This implementation was efficient for single core processors. However, on a chip-multiprocessor [14], where real concurrency is available, this single packet pool became the bottleneck. Therefore, the single reader/writer queues have been implemented.

3.4 Zero Copy Stack

As Java is a safe language there is no need to distinguish between user and kernel mode in the TCP/IP stack. Memory access is safe due to strong typing and runtime checks (e.g., array bound checks). Therefore, packet data is never copied within ejIP. A packet buffer is filled on reception by the link layer and the buffer is forwarded through the layers. A send packet is filled by the application and forwarded towards the link layer for transmission. Very simple applications, e.g., ping reply in ICMP, just respond with a packet

```
new RtThread(NET_PRIORITY, NET_PERIOD) {
        public void run() {
                for (;;) {
                        waitForNextPeriod();
                        net.run();
                }
        }
};
```

Figure 2. Using JOPs real-time thread for the periodic Net loop.

on a received package. Those applications can reuse the received packet buffer for the reply.

3.5 Single and Multithreaded Usage

Several classes are *active* classes where a method has to be executed periodically. These active functions can be invoked from a single thread or can be distributed to several threads or processor cores. The active classes are: the link layer, Net, Udp, and Tcp.

On JOP a simple real-time thread system is available. Each thread is represented by RtThread and shall implement an infinite loop within run() with a call to waitForNextRelease() to suspend the thread. The thread is resumed again after its period. This is similar to periodic threads in the RTSJ [1]. Figure 2 shows the usage of an RtThread for the periodic invocation of run().

On a plain Java system this periodic behavior can be simulated with a call of sleep(). For a single threaded usage of ejIP a single loop can be used to invoke all relevant methods. As ejIP uses polling and handler based distribution of packets, the TCP/IP application is automatically included.

3.6 Real-Time Systems

Real-time threads (or tasks) are usually organized as periodic threads. Each thread has a period and a priority assigned according the period (or deadline). Fixed priority, preemptive schedulers are used for task scheduling. To make WCET and schedulability analysis feasible threads shall not block. The BSD socket based API is based on blocking calls for read and write operations and therefore not the first choice for an analyzable real-time system.

The organization of ejIP into periodic tasks fits well for real-time Java systems, such as the RTSJ [1] or SCJ [8]. All parts of ejIP are organized as periodic, polling tasks. No blocking can occur. The code contains only bounded loops and is WCET analyzable. We have used ejIP as an example to test the WCET analysis tool for JOP [21].

4. Application Program Interface and Usage

ejIP provides its own API for network programming instead of using the standard Java, socket based network API. The reasons for this are twofold: (1) the handler based API of ejIP uses less resources and can even be used without any threading support of the JVM; (2) avoiding blocking read or write operations enables usage of ejIP in real-time systems.

An UDP or TCP application has to implement a handler class. The handler contains a method that is invoked on a packet receive. Furthermore, each handler contains a method that is periodically invoked for any time related processing.

Figure 3 shows the abstract class of a TCP handler that an application needs to implement. The TcpHandler class contains also the reference to the object that represents the TCP connection. Two different methods are invoked with the receiving packet as parameter: established() when a new connection is established and request() on a received packet otherwise. Both methods return

```
public abstract class TcpHandler implements Runnable {

        /**
         * The connection we are handling
         */
        protected TcpConnection connection;

        /**
         * Connection is established. Transfer can start.
         */
        public abstract Packet established(Packet p);

        /**
        * handle one request on the registered port.
        */
        public abstract Packet request(Packet p);

        /**
         * Close connection
         */
        public abstract boolean finished();

        /**
         * Application logic that gets invoked periodically
         */
        public void run() {
        }
}
```

Figure 3. The TCP handler interface

a packet to be sent back. The application can reuse the packet buffer of the received packet for the reply. On a connection close the method finished() is invoked. The run() method is invoked periodically and can be overwritten by the TCP application to perform time related functions. Within the run() methods new TCP packets can be generated and enqueued for transmission.

4.1 Usage Example

As a usage example Figure 4 shows a very simplified Telnet demon. It listens on port 23 for a connection and understands one *command*: hello. On connection establishment a greeting message is sent back. The received packet is just reused and the greeting message is set with a utility function of Ip. Within request() depending on the input message (the command hello is recognized) either the Hello message or an empty TCP package is sent back. This example also shows usage of ejIP in a single thread. Within an infinite loop in main() the link layer and the network stack are invoked.

5. Evaluation

The best proof of the usefulness of a software package is usage in real applications. The TCP/IP stack ejIP is part of the distribution of JOP and in use in several industrial applications [17]. In the following section we describe some projects that we are aware of that use ejIP.

5.1 The SCADA Device TeleAlarm

TeleAlarm (TAL) is a typical remote terminal unit of a supervisory control and data acquisition (SCADA) system. It is used by the Lower Austria's energy provider EVN (electricity, gas, and heating) to monitor the distribution plant and to remotely control gas valves. The TAL contains several EMC protected digital input and output

```
public class Telnetd extends TcpHandler {

        static Net net;
        static LinkLayer ipLink;
        StringBuffer sb = new StringBuffer();
        StringBuffer cmd = new StringBuffer();

        public static void main(String[] args) {
                Ejip ejip = new Ejip();
                net = new Net(ejip);
                int[] eth = {0x00, 0xe0, 0x98, 0x33, 0xb0, 0xf8};
                int ip = Ejip.makeIp(192, 168, 0, 123);
                ipLink = new CS8900(ejip, eth, ip);
                net.getTcp().addHandler(23, new Telnetd());

                for (;;) {
                        ipLink.run();
                        net.run();
                }
        }

        public Packet request(Packet p) {

                StringBuffer hello =
                        new StringBuffer("Hello from JOP\r\n");
                Ip.getData(p, Tcp.DATA, sb);
                StringBuffer resp = null;
                if (sb.length()!=0) {
                        for (int i=0; i<sb.length(); ++i) {
                                char ch = sb.charAt(i);
                                if (ch!='\n' && ch!='\r') {
                                        cmd.append(ch);
                                } else {
                                        String s = cmd.toString();
                                        if (s.equals("hello")) {
                                                resp = hello;
                                        }
                                        cmd.setLength(0);
                                }
                        }
                }
                if (resp!=null) {
                        Ip.setData(p, Tcp.DATA, resp);
                } else {
                        p.len = Tcp.DATA<<2;
                }
                return p;
        }

        public Packet established(Packet p) {
                Ip.setData(p, Tcp.DATA, "Welcome to JOP\r\n");
                return p;
        }

        public boolean finished() {
                return false;
        }
}
```

Figure 4. Implementation of a minimal Telnetd example with the TcpHandler

ports, two 20 mA input ports, Ethernet connection, and a serial interface. Furthermore, the device performs loading of a rechargeable battery to survive power down failures. On power down, an important event for a energy provider, an alarm is sent.

The communication between the TAL and the main supervisory control system is performed with a proprietary protocol. On a value change TAL sends the new data to the central system. Furthermore, the remote units are polled by the central system at a regular base. The TAL itself also sends the actual state regularly. TAL can communicate via Internet/Ethernet, a modem, and via SMS to a mobile phone. The IP based version of the proprietary SCADA protocol uses UDP for communication. A simple web server displays the status of the digital and analog inputs. For safety and security reason there is no connection between the control network and the office network or the Internet.

The whole system is written in Java and uses JOP as execution platform. The UDP based communication and the web server uses an early version of ejIP.

5.2 Support for Single Track Railway Control

Another application of ejIP is in a communication device to support single-track lines of the Austrian Railways (ÖBB). The system helps the superintendent at the railway station to keep track of all trains on the track. He can submit commands to the engine drivers of the individual trains. Furthermore, the device checks the current position of the train and generates an alarm when the train enters a track segment without a clearance.

At the central station all track segments are administered and controlled. When a train enters a non-allowed segment all trains nearby are warned automatically. This warning generates an alarm at the locomotive and the engine driver has to perform an emergency stop.

Figure 5 gives an overview of the system. The display and command terminal at the railway station is connected to the Intranet of the railway company. On the right side of the figure a picture of the terminal that is connected to the Internet via GPRS and to a GPS receiver is shown. Each locomotive that enters the track is equipped with either one or two of those terminals.

The current position of the train is measured with GPS and the current track segment is calculated. The number of this segment is regularly sent to the central station. To increase the accuracy of the position, differential GPS correction data is transmitted to the terminal. The differential GPS data is generated by a ground base reference located at the central station.

The exchange of positions, commands, and alarm messages is performed via a public mobile phone network (via GPRS). The connection is secured via a virtual private network that is routed by the mobile network provider to the railway company's Intranet. The application protocol is command/response and uses UDP/IP as transport layer. Both systems (the central server and the terminal) can initiate a command. The system that sends the command is responsible for retries when no response arrives. The deadline for the communication of important messages is in the range of several seconds. After several non-successful retries the operator is informed about the communication error. He is then in charge to perform the necessary actions.

Besides the application specific protocol based on UDP, a TFTP server is implemented in the terminal. It is used to update the track data for the position detection and to upload a new version of the software. The flexibility of the FPGA and an Internet connection to the embedded system allows upgrading the software and even the processor in the field.

5.3 A Gyrotron Controller

Michel and Sachtleben at the Max-Planck-Institut für Plasmaphysik developed a Gyrotron controller based on JOP [13]. The Gyroton controller is used within the Wendelstein 7-X project, a research project on controlled nuclear fusion reaction. The Gyrotron controller implements time-critical control functions in hardware in an FPGA. The same FPGA also contains the soft-core processor JOP, which reads and writes parameters for the hardware controller. The values are received via an Ethernet connection. The whole application is written in Java and ejIP is used for the network stack.

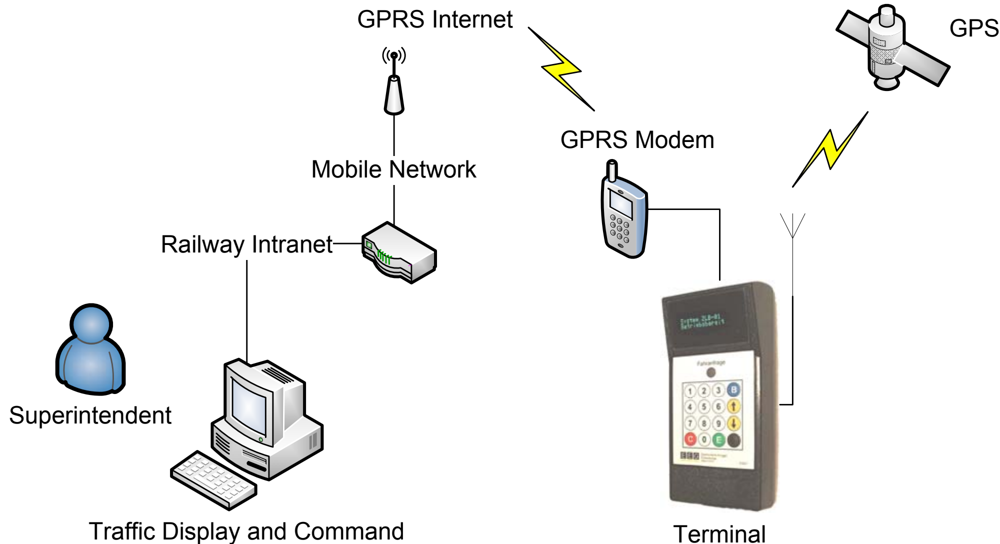

Figure 5. Communication in the support system for single-track railway control for the Austrian railway company

5.4 Lessions Learned

The three described projects show applications from very different domains that use and rely on ejIP to handle the communication. Two aspects are common to all systems: soft real-time requirements can be better achieved via UDP and update of the embedded software is possible with TFTP.

The unusual polling model, instead of a stream based interface, of ejIP fits quite well for this kind of embedded applications. The ejIP interface was actually designed as a result of the embedded applications. Furthermore, the polling (non-blocking) interface to ejIP enables TCP/IP in single threaded environment. Therefore, very small JVMs, which support only a single thread of control, can use ejIP.

5.4.1 Retransmission in UDP instead of using TCP

The transport layer, TCP in the case of the Internet, usually provides reliable communication. However, the timeouts in TCP are way longer than the communication deadlines within control systems. The approach in all three presented projects is to use a datagram-oriented protocol and to perform the timeout and retransmission at the application level. To simplify the timeout handling a simple command and response pattern is used. One partner sends a command and expects the response within a specific time bound. The command initiator is responsible for retransmission after the timeout. The response partner just needs to reply to the command and does not need to remember the state of the communication. After several timeouts the communication error is handled by an upper layer.

5.4.2 Software Update

Correction of implementation bugs during development can be very costly when physical access to the embedded system is necessary for a software update. Furthermore, a system is usually never really finished. When the system is in use, the customer often finds new ways to enhance the system and requests additional features.

Therefore, an important feature of a networked embedded system is a software and parameter update in the field. The described projects use the Internet protocol for communication and therefore TFTP is a natural choice. TFTP is a very simple protocol that can be implemented within about 100 lines of code. It is applicable even in very small and resource constraint embedded devices.

It has to be mentioned that no advanced process is implemented to update the software during runtime [23]. In the railway application, the new software version is downloaded in the background and stored in a NAND Flash. On the next start of the device the software is copied from the NAND Flash to the regular Flash and the device reboots with the new software. The background download is implemented to avoid long startup times of the device when a new software version is available.

6. Conclusion

In this paper we presented a small TCP/IP stack, named ejIP, for embedded Java. It is implemented in Java and represents an example of using Java for tasks that are usually implemented within an operating system. Our overall goal is to implement a Java runtime system that does not need an operations system and can therefore be implemented in a single language – Java. Furthermore, ejIP is optimized for minimum resource consumption and usage in real-time systems. An example application with a small web server consumes 76 KB for the runtime system. The handler based application interface enables non-blocking execution of the TCP/IP stack, which is essential for real-time tasks.

A. Source Access

The source of ejIP is open-source released under the BSD license (similar to the original BSD Unix source on which most TCP/IP implementations are based on). The source is part of the JOP distribution. Download information can be found at `http://www.jopwiki.com/Download`. The Java source of ejIP is located under java/target/src/common/ejip.

Acknowledgments

I would like to thank Wolfgang Puffitsch for enhancing the TCP part of ejIP.

References

[1] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java

Series. Addison-Wesley, June 2000.

[2] Mei-Ling Chiang and Yun-Chen Li. LyraNET: A zero-copy TCP/IP protocol stack for embedded systems. 2006.

[3] Adam Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM Press.

[4] Adam Dunkels, Juan Alonso, and Thiemo Voigt. Making TCP/IP viable for wireless sensor. Technical report, November 18 2003.

[5] Adam Dunkels, Juan Alonso, Thiemo Voigt, Hartmut Ritter, and Jochen H. Schiller. Connecting wireless sensornets with TCP/IP networks. In Peter Langendörfer, Mingyan Liu, Ibrahim Matta, and Vassilios Tsaoussidis, editors, *WWIC*, volume 2957 of *Lecture Notes in Computer Science*, pages 143–152. Springer, 2004.

[6] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[7] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. Making sensor networks IPv6 ready. In Tarek F. Abdelzaher, Margaret Martonosi, and Adam Wolisz, editors, *SenSys*, pages 421–422. ACM, 2008.

[8] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[9] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research (MSR), October 2005.

[10] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. IEEE, 2001.

[11] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[12] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[13] Georg Michel and Jürgen Sachtleben. An integrated gyrotron controller. *Fusion Engineering and Design*, In Press, Corrected Proof:–, 2011.

[14] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[15] Nilesh Rajbharti. Microchip TCP/IP stack. Application Note AN833, 2002.

[16] Joel J. P. C. Rodrigues and Paulo A. C. S. Neves. A survey on IP-based wireless sensor network solutions. *Int. J. Communication Systems*, 23(8):963–981, 2010.

[17] Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.

[18] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[19] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted 2009, 2011.

[20] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[21] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[22] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[23] Suriya Subramanian, Michael W. Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 1–12. ACM, 2009.

[24] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[25] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[26] In-Su Yoon, Sang-Hwa Chung, and Jeong-Soo Kim. Implementation of lightweight TCP/IP for small, wireless embedded systems. In Irfan Awan, Muhammad Younas, Takahiro Hara, and Arjan Durresi, editors, *AINA*, pages 965–970. IEEE Computer Society, 2009.