IMPLEMENTING NETWORK PROTOCOLS IN JAVA - A FRAMEWORK FOR RAPID PROTOTYPING

Matthias Jung, Ernst W. Biersack
Institut Eurecom, 06904 Sophia-Antipolis, France,{jung,erbi}@eurecom.fr
Alexander Pilger
Siemens AG, 81730 Munich, Germany, alexander.pilger@mchp.siemens.de

In Proceedings of ICEIS'99, Estubal, Portugal

Keywords: protocol implementation, rapid prototyping, java, portability, configurability, module

Abstract: This paper presents JChannels,[*] a framework to support the implementation of network protocols in Java. The goals of JChannels are the rapid development of structured, reusable, and configurable protocol stacks profiting from Java features like incorporate concurrency, portability, and runtime class loading. We present the JChannels architecture show how to work with JChannels, give an example implementation of a simple transport protocol, and provide some performance results.

1 MOTIVATION

The communication requirements of upcoming distributed applications foster the development of new specialized communication protocols. Unfortunately, protocol implementations are software of high complexity. Implementing new protocols from scratch is therefore a difficult and cumbersome task. Protocol frameworks are meant to simplify protocol development by providing libraries supporting the basic functions of protocols and imposing a structure on protocols to support reuse and maintainability of protocols. Besides, they are a good playground for testing new protocols.

Various protocol frameworks already have been developed. Each one has a special focus and different features. The best known frameworks are X-Kernel (Hutchinson & Peterson 1991) and STREAMS (Unix 1990) both residing in the kernel of the operating system, or Conduits+ (Hüni, Johnson & Engel 1995) running in user-space.

All these systems suffer from their lack of portability. This made us examine if Java (Sun Microsystems 1995) is suitable for protocol implementation. Besides being highly portable, Java supports modular programming in an object oriented fashion, allows for asynchronous thread handling, and comes along with a comfortable library. Wherever Java is supported, communication protocols written in Java may profit from Java protocol development frameworks. This may apply in the future also for Embedded Java Systems and Java Cards.

Protocols implemented in Java running inside the Java Virtual Machine (JVM) on top of an operating system (OS) suffer from a performance penalty compared to protocols implemented in the OS kernel. However, we think that in many situations performance is not the primary goal. For rapid prototyping, testing, and monitoring it is more important that protocols can be modified, extended and configured easily.

HotLava (Krupczak, Calvert & Ammar 1998) is one of the first protocol frameworks in Java. It basically served to evaluate the performance of a special protocol suite in Java compared with a C implementation without considering re-use and flexibility. iBus (Maffeis 1997) is a Java framework that is meant to mainly support multicast

[*]For more information see the JChannels Homepage http://alpes.eurecom.fr/~projava

communication protocols on transport level.

The re-implementation of the Conduits+ framework (Hüni et al. 1995) in Java is the work closest to JChannels with regard to the design goal of re-usability. However, it differs widely with regard to the architectural approach: Conduits+ heavily makes use of design patterns to re-use the infrastructure of protocols. It seems to transfer the complexity of network protocols into the framework instead of simplifying implementation.

JChannels is the first completely portable framework that reduces the complexity of protocol implementation to support rapid prototyping of structured, re-usable and flexible protocol stacks for any kind of communication protocols.

2 DESIGN

2.1 Design goals

JChannels is not only a framework to implement and run communication protocols in user space. It is first of all a proposal to structure protocol implementations. Protocols developed with JChannels have the following attributes:

- Rapid prototyping: module programming is simplified by providing communication specific classes. Once protocol modules are programmed, it is easy to link them to a new protocol stack without major efforts.

- Re-usability: protocols are divided in small parts (modules), which can be used in different protocol stacks.

- Maintainability: protocols are obliged to be programmed in a way that they can be easily maintained, modified, and extended.

- Flexibility: it is possible to customize and configure protocols following the specific needs of applications[1].

- Portability: JChannels profits thereby from the Java concept of the virtual machine, i.e. an interpreter of machine independent Java code.

JChannels claims to support the implementation of all kind of network protocols up from the network layer and provides a generic interface to access lower layers.

2.2 JChannels Overview

JChannels provides a set of classes to guide the programmer through implementation. We thereby profit of Java's **object-oriented design** principles of inheritance, data encapsulation and polymorphism. A protocol suite is represented by the class `Stack` and consists of a set of protocol functions implemented each in the class `ProtocolModule` (e.g. Error Control, Flow Control, Fragmentation, etc). The class `Message` represents the messages exchanged between protocol entities and is responsible for the validity of the data. The programmer must inherit new messages, modules, and stacks from their super classes.

In order to simplify programming, JChannels provides a **library** of classes often needed for protocol implementation. The class `ProtocolGraph` helps to build new stacks out of modules, the class `Timer` allows modules to use asynchronous timer. Other examples are `SeqNr` (access to sequence numbers), `Buffer` (buffer handling) or `Data` (manipulation of arrays).

A transparent **sub-system** assures that the programmer does not need to care about how messages are transported. He is only concerned about the correctness of modules and stack. This subsystem follows the **thread-per-message** paradigm, i.e. each message arriving gets its own thread, which calls all modules one after the other with the message as the argument (see Figure 1). A thread handler controls the number of threads running at the same time. This concept

[1]Note that tailoring in JChannels (the dynamic modification of protocol graphs at run-time) is possible in principle due to the Java facility of dynamic class loading. But it is not supported explicitly by JChannels, i.e. there are no mechanisms of controlling inter-operability and providing security

reduces context switches and has shown to be more efficient than a thread-per-module model (Hutchinson & Peterson 1991). JChannels provides a generic `NetDriver` interface to hide implementation details of the network access to the programmer.

When multiple applications share the same input port, de-multiplexing is necessary, which has shown to be costly (Tennenhouse 1989). We introduced a new design element called **network anchor**, where all demultiplexing functionality is concentrated (see Figure 2). That way, we ensure that every application has its own stack entity (Roca, Braun & Diot 1997). A network anchor can be seen as a daemon process within a small network operating system where applications can register a new stack while running as an independent process. Java's Remote Method Invocation (Sun Microsystems 1999) is used for inter-process communication between application and stack. An anchor is only needed when applications share a service access point. If an application is running above a TCP socket, for example, they can run a stack in **stand-alone-mode**, which means it has direct access and runs it in the same process.

In order to re-use protocol modules in different contexts they must implement only a single function that is part of a more complex protocol and they must be autonomous. Autonomy is guaranteed in JChannels by using an **event model** for communicating control calls between modules. The stack thereby serves as the event handler and is responsible to map events of one module to the right function of another (e.g. AckArrivedEvent, BufferFullEvent). Events may also change global state variables in the stack.

2.3 JChannels classes

2.3.1 The classes Message, MessageThread, StackManager

The class `MessageThread` represents an own thread within a Java virtual machine escorting an instance of the class `Message` through the

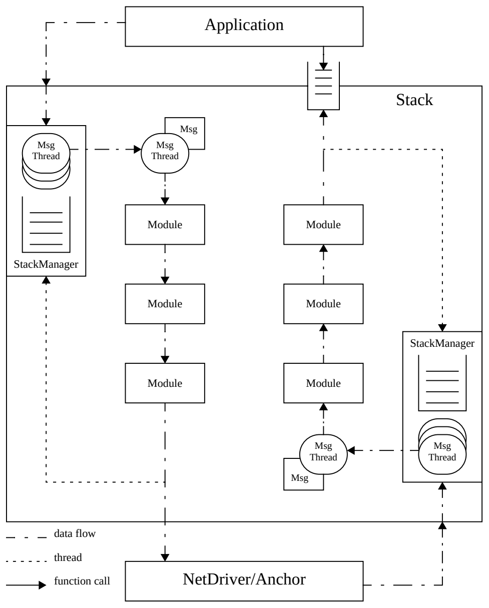

Figure 1: Thread handling and message traversal

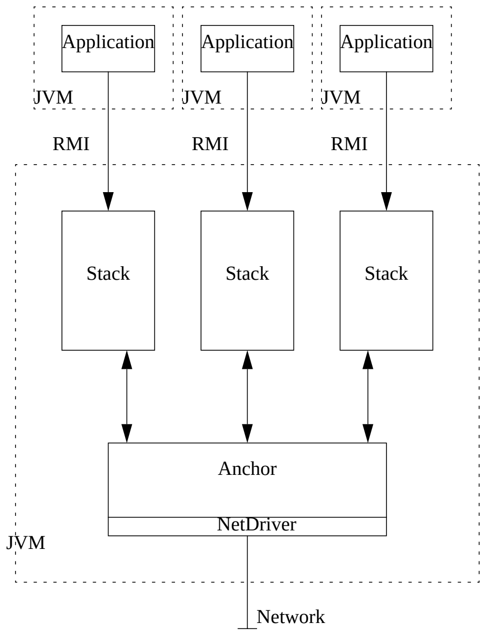

Figure 2: The JChannels anchor system

protocol graph by calling one module after the other. A `protocol graph` is a linear list of protocol modules defining an order. By default, every stack contains two protocol graphs, one defining the way from application to network, the other from network to application. The main idea of a **stack manager** is to limit the number of running message threads[2]. Message threads can only be created and started upon requesting the `StackManager`[3]. If the current thread is running, the request is put in a queue and processed as soon as the thread returns. Protocol modules can demand a priority for a thread request. When a message thread is no longer used, a protocol module should give it back to the stack manager in order to make it available for other threads.

2.3.2 The class ProtocolModule

The idea of a protocol module is to implement a part of a protocol, ideally just a single function. The core of each module is represented by the method `callModule()` that is the entrance point for all message threads to process their data[4]. Protocol modules don't know other modules and therefore don't call methods of other modules. This avoids dependencies and improves the re-usability of modules. Instead, **event** driven indirect communication is used: A protocol module informs the stack about an event, which causes the stack to call other modules that are concerned by this event. In order to indicate an event, the method `throwEvent()` calls the method `Stack.eventCall()` and hands information in form of an instance of the class `Event` to the stack. This information can be used as parameter for other modules or global state information. Every module can define its own events, the stack is responsible to process them in the right way. For See Figure 3 for how events are triggered.

[2]A stack manager can manage more than one message thread. However, on a single CPU computer there is no value to run several message threads in parallel. For our system, only one single thread per thread manager is running at the same time

[3]There are two stack managers per stack, one for messages from the application, one for messages from the network.

[4]Note that protocol modules have by default only one entrance point for a message thread. There is no up-call or down-call as it would be for a more layer oriented approach

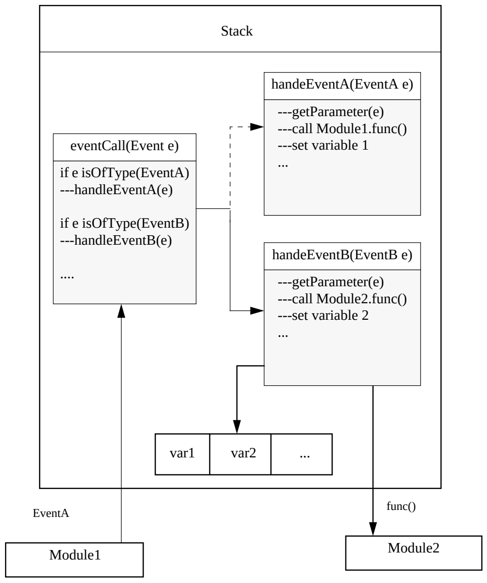

Figure 3: Event handling

In order to use timers, a protocol module must implement the abstract method `timerCall()` of the interface `TimerCompatible`.

2.3.3 The class Stack

A stack is instantiated either by an `Anchor` within the JChannels anchor system (see below) or as a stand-alone-stack by an application. The stack has to main tasks:

First, it acts as a *glue* to put together the various independent modules in order to let them work together as an integral whole. Therefore, the stack instantiates all necessary modules and builds the protocol graph. In the method `eventCall()` it implements an event dispatcher that calls methods to handle all events that are indicated by the various modules by calling the right method of another module or updating global state variables. See Figure 3 to get an impression about how event handling works.

Second, it serves as an *interface* for application and network. If the stack should work without an anchor, it is started as an own thread looking if data is arriving at the network. The application has access to all public functions of the stack. When a stack works within a system with an anchor, network access becomes the task of the anchor and no stack thread is needed. In the latter case, when application and anchor/stack are running in different Java virtual machines, the mechanism of remote method invocation (RMI) is used to give the application access to the stack. The main standard methods of interactivity between stack and application are the functions `acceptPacket()` and `deliverPacket()` to write and read packets, respectively. For data exchange with the network, the stack provides the function `acceptFrame()` to receive messages from the network anchor. Besides its methods for application and network, each stack contains instances of special modules called *proxy* modules to write a message to the application output queue or to the network by calling the anchor or the network driver to write it physically to the network.

2.3.4 The class Anchor

Besides providing network access, the network `Anchor` class has two main tasks.

The first task is to *register* and *de-register* stacks demanded by an application, which means loading their code in the virtual machine and instantiate them. JChannels hereby profits from the Java facility of dynamic class loading at runtime. An application registers a stack by sending data over a well known port that is observed by an instance of the class `RegDriver`. The registration message is containing information about the name of the stack and the keyword[5] to access the stack as a remote object. The registration driver is called periodically by the anchor to see if a registration message arrived. De-registration is performed by the stack itself when it will be deactivated.

The second task is to demultiplex incoming messages to the right stack instance. Each stack that has registered must indicate its own address mask applied for demultiplexing. For that purpose, the class `AddressMask` is provided. An address mask consists of two byte arrays: the first array specifies positions of bytes within an incoming message, the second contains the value that these bytes should have. A special mechanism guarantees that a stack is started only when its address mask is unique and unambiguous.

[5]see Java RMI specification

2.3.5 The class NetDriver

Messages are read from and written to the network by calling an object of the class `NetDriver`. The purpose of this class is to run the anchor or the stack on different platforms without changing its functionality. Anchor or stack do not know if its network drivers are running over UDP, TCP, AppleTalk, Ethernet or Token-Ring. They just use the drivers given to access the network.

2.3.6 The class TimerPool

To handle many timers at the same time without creating a new thread each time a timer is required, a global `Timerpool` is used. This TimerPool is accessible for all subclasses of a protocol module. Each module using timers needs to implement the method `timerCall()` of the interface `TimerCompatible`.

3 WORKING WITH JCHANNELS

We briefly describe what must be done to implement a protocol in JChannels and present an example protocol realized with JChannels.

The main steps to implement a protocol in JChannels are

- Analyze a protocol and identify all functions to be performed and map each function to a module

- Distinguish if the modules are processing data in direction from application to network (output direction) or from network to application (input direction)

- Build the protocol graph for both directions by connecting the modules to unidirectional chains
- Identify the information that every module needs and the information that every module provides
- Map the relation between information source and sink by events and implement an event handler in the stack class
- Make sure that the stack defines a valid address mask and terminates correctly

As a first test for JChannels, we implemented a connection oriented transport protocol that guarantees reliable data transfer. A server application is waiting for connection requests and sends a file to the client. In the stack on the server side, packets taken from the application are cut to a configurable size.

The protocol stack for the **server** consists of the following modules. We start with the modules working in the **output direction**. A `Fragmentation` module forwards incoming messages when they are smaller than the size configured by the stack. Larger messages are split into two or more messages. An `Addressing` module adds the destination address to the message header. The destination address must be handed to the module by the stack. A module `Retransmission` adds a sequence number to the header, buffers every passing message, and starts a timer for each message sent. When the timer expires the message is retransmitted. Acknowledgments indicated by the stack lead to freeing the retransmission buffer. A `ConnectionOutput` module is conceived to send control messages that concern the connection (requests,confirm,error). For testing we added two modules `ErrorSimulation` and `Logging` that can be switched off.

For the **input direction** the server stack consists of the following modules. A `ConnectionInput` module is conceived to process incoming connection control messages and indicate the stack the corresponding events (destination address changed, connection opened/closed, error, ...). A module `AckHandler` identifies incoming acknowledgments and informs the stack, which then informs the Retransmission Module.

The stack of the **client** consists for the **input direction** of the following modules: `ConnectionInput` is the same module as for the server (is conceived to work for server and client). `Reassembly` manages the receive buffer. A message is thrown away when it does not fit in the buffer. The message is buffered when it is not the next message expected (sequence number), but fits into the buffer. The message is forwarded to the neighbour when it is the right message. In the case that a buffered message succeed the last delivered message, this message is delivered, too. Besides, this module signals events to the stack that contain buffer information and indicate if the message could be delivered, had to be buffered, or must have been thrown away. A module `RemoveAddress` strips off the address header, i.e. it just removes the data bytes in the header that represents the address of its own stack.

For the **output direction** we have a `Feedback` module sending acknowledgments upon arriving data messages and another `ConnectionOutput` module as for the server.

This is only a simple example for a transport protocol. We recently implemented a complete TCP plus the IP functionality that is needed on end-to-end basis. The complete design and implementation process was accomplished in less than three man-months. We therefore claim that even the implementation of highly complex protocols is not only possible but also extremely simplified following the modular design principles of JChannels.

4 DISCUSSION

Our design principle of autonomous modules has some implications that must be payed attention to. A module does not know anything about another module. It is therefore not able to make demultiplexing decisions. Processing of an incoming[6] message therefore will happen as follows: the module checks if the message is destined for it and either processes the message (when it is destined for the module) or forwards it to its neighbour otherwise. A module should throw a message away only when it can identify it (e.g. an acknowledgement in input direction that should not any longer be forwarded after it triggered a buffer deallocation). Another approach would be to extend the message class by functions to check the correctness of the data that will be processed.

The access to the message information can be made more comfortable by providing conversion functions from byte arrays to integers, floats or even more complex data types. The message could even define fields (like sequence number, source address, etc.) and provide functions to access them. That way, message manipulations described in Section 2 can be widely replaced. A message then would be responsible for assigning a byte array correctly to its defined fields and to transform all fields back into a byte array.

Since no demultiplexing within the stack is done, the protocol graph consists of an unidirectional chain of modules for each direction. Nevertheless is it possible to support multiple protocol graphs, which would imply an additional de-multiplexing step performed by the stack.

In order to allow protocol modules to be used in different contexts, i.e. different protocol stacks with different header formats, information about the relevant data in the message (relevant header fields, length of the payload, or flags) should be given to the protocol modules at instantiation time or even at runtime by the stack.

[6]as **output** direction we refer to a message going from application to network, as **input** direction we refer to a message coming from the network

In our model, each connection is represented by an own stack. In our example stack, connection state is handled by special modules. An alternative approach would be to run a connection server which registers new stacks at the network anchor.

Clark and Tennenhouse identified Application Level Framing (ALF) as a key architectural principle and Integrated Layer Processing (ILP) as a key engineering principle of modern communication protocols (Clark & Tennenhouse 1990). JChannels confirms with application level framing since it allows the application to configure easily all communication parameters and to specify the data unit given to the stack. By contrast, ILP is not only not supported by JChannels, it contradicts the whole design philosophy. Clarc/Tennenhouse identified as possible drawbacks of ILP the *complex design that may complicate maintainability and overall utility of protocol software*. We aim at doing exactly the opposite.

5 PERFORMANCE

To get an idea about the performance of JChannels, we made a simple measurement using the stack described above running it stand-alone (without an anchor system and RMI) over a UDP socket. We measured the total transmission time of a file of size 1 Megabyte using Sparc Ultras 1 over a 10MBit Ethernet. The highest throughput achieved for a file of 1 Megabyte was 60 KByte/s. The same protocol implemented in a monolithic C program reached a throughput of 800 KByte/s. That means, the performance penalty due to Java and modularization is around a factor of 10-15.

A further analysis showed that using a just in time compiler (JIT) could speed up the performance by approximately 30 percent. This is not very much since JITs have a potential to speed up Java programs by a factor of up to 25. The frequent creation of new objects and threads provokes the Java garbage collector to free variable space. Since garbage collection is already optimized for the interpreter, the performance gain of the JIT stays rather low. The reuse of threads and a message object pool may speed up performance significantly, especially for compiled Java code.

6 CONCLUSION

JChannels is a protocol environment conceived to support rapid and structured protocol development by producing reusable and maintainable code. We propose to take advantage of features of the object-oriented programming language Java to improve and refine existing protocol environments. Java comes along with a comfortable library, provides an easy way to integrate code during runtime, supports thread handling, and is platform independent.

The focus of JChannels is on good structuring of protocols that allows to reuse, extend and configure protocols easily. We therefore replaced the conventional layer model and defined protocol modules by a smaller unit of functionality that does not use up- and down-call, but just processes incoming data, accesses resources, and sets state variables accordingly. In order to allow a protocol module to be reused in different contexts (stacks), an event model was introduced that avoids direct communication between modules, and de-multiplexing was concentrated on the lowest level.

Development of protocols is considerably simplified by JChannels. The implementation of TCP in less than three man months shows that JChannels is able to cope with complex protocols and is well suited for rapid prototyping, testing and monitoring of network protocols.

Acknowledgment

The authors like to thank Stephan Röösli and Marie-Noëlle Sauvayre who were involved in the design and implementation process of JChannels.

This work was sponsored by Siemens, Munich.

References

Clark, D. D. & Tennenhouse, D. L. (1990), Architectural considerations for a new generation of protocols, *in* 'Proc. ACM SIGCOMM 90', Phildelphia, PA, pp. 200–208.

Hüni, H., Johnson, R. & Engel, R. (1995), A framework for network protocol software, *in* 'Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)', ACM Press.

Hutchinson, N. & Peterson, L. (1991), 'The x-kernel: an architecture for implementing network protocols', *IEEE Transactions on Software Engineering* **17**(1), 64–76.

Krupczak, B., Calvert, K. & Ammar, M. (1998), Implementing protocols in java: The price of portability, *in* 'IEEE Infocom '98'.

Maffeis, S. (1997), ibus - the java intranet software bus.

Roca, V., Braun, T. & Diot, C. (1997), 'Demultiplexed architectures: a solution for efficient streams based communication stacks', *IEEE Networks Magazine*.

Sun Microsystems (1995), The java virtual machine specification, Technical report.

Sun Microsystems (1999), 'The java remote method invocation specification'. http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-spec.ps.

Tennenhouse, D. L. (1989), Layered multiplexing considered harmful, *in* H. Rudin & R. Williamson, eds, 'Proc. IFIP Workshop on Protocols for High-Speed Networks', North-Holland Publ., Amsterdam, The Netherlands, Zurich, Switzerland, pp. 143–148.

Unix (1990), 'Streams programmer's guide', *Unix System V Release 4*.