IST Project IST-2001-32329 HIDOORS

Deliverable 4.2
Remote Event Service Design

Edited by
Marc Schanne
Dr. James J. Hunt
Forschungszentrum Informatik
Karlsruhe, Germany

Contents

1 Event Channel Network **5**

1.1 Requirements 5

1.2 Comparable Development 6

1.3 Architecture 7

2 Socket Foundation **9**

3 Logical communication channels **11**

4 Event Handling **15**

4.1 Events 15

4.2 Handling Infrastructure and Process 16

4.3 Queues 18

4.4 Administration Channel 19

4.5 Miscellaneous Utility Classes 21

5 XSL Transformation **22**

6 Failure Modes and Error Handling **24**

6.1 UDP/IP over Ethernet 28

6.1.1 Physical Layer 29

6.1.2 Data Link Layer 30

6.1.3 Network Layer 31

6.1.4 Transport Layer 32

6.2 CAN 32

6.2.1 Physical Layer 33

6.2.2 Data Link Layer 34

6.2.3 Network Layer 36

6.2.4 Transport Layer 36

6.3 TTP/C . . . 37
6.3.1 Physical Layer . . . 37
6.3.2 Data Link Layer . . . 38
6.3.3 Network Layer . . . 39
6.3.4 Transport Layer . . . 39
6.4 Event Channel Network . . . 42
6.4.1 Session Layer . . . 42
6.4.2 Presentation Layer . . . 44
6.5 Application Layer . . . 45
6.5.1 UDP/IP over Ethernet . . . 45
6.5.2 Controller Area Network (CAN) . . . 45
6.5.3 Time-Triggered Protocol, SAE class C (TTP/C) . . . 46
6.6 Summary . . . 46

7 Testing **47**
7.1 Test programs . . . 47
7.1.1 Simple Server and Clients . . . 48
7.1.2 Different Servers a ClientServer and a Client . . . 49
7.1.3 Ping Server and Client . . . 50
7.1.4 Timeout Example . . . 51
7.1.5 Dynamic Generation . . . 51
7.2 Application Implementation . . . 52
7.2.1 Structure of the programs . . . 53
7.3 Implementation of XSLT based applications . . . 53
7.3.1 XML and XSL Transformation Example . . . 54

8 Conclusion **55**

A Structure of the API **56**

B Diagrams and XML/XSLT **58**
B.1 Main classes and associations . . . 58
B.2 DTD for the description of a node . . . 58
B.3 XSLT file for simple client/server nodes . . . 62
B.4 Example XML files . . . 70

Bibliography **73**

Foreword

This document is the deliverable D4.2 describing the **HIDOORS** remote event service. It is the conclusion of task 4.7.

Version	Remark	Updated by	Date
0.1	Initial Version	Marc Schanne	11.2002
0.2	Editing	Dr. James J. Hunt	27.1.2003
0.3	Editing	Marc Schanne	27.1.2003
0.9	Editing	Marc Schanne	28.08.2003
0.95	Editing	Marc Schanne	23.01.2004
1.0	Editing	Dr. James J. Hunt	17.02.2004
1.1	Reorganization	Marc Schanne	15.03.2004
1.2	Edit Error Handling Chapter	Dr. James J. Hunt	28.04.2004
1.3	Edit Rest of Document	Dr. James J. Hunt	28.04.2004

Chapter 1

Event Channel Network

The event channel network is designed for component and node communication in a distributed application in the **HIDOORS** project. It provides a network independent means of remote event transmission between Java Virtual Machines connected directly and/or over a realtime bus. The Programming API is based on 100% pure Java with RTSJ extensions.

The event service support two main models of communication: the receiver monitoring and sender monitoring. Receiver monitoring can be used for systems with a know event rate. In this mode, the receiver must react when events fail to arrive. On example of receiver monitoring is an overall system health and recovery monitor that receives events from all parts of a system and takes action when the system falls outside of desired operational parameters. Sender monitoring can be used in a control and monitor system, where the sender sends control signals to some part of the system and then monitors status information to insure that the proper action occurred. Both models can be used across a juncture between realtime and non-realtime components.

To enable the communication between nodes in the event channel network, a lightweight protocol is used to transmit events with low latency. Using an event paradigm decouples senders and receivers, simplifying dynamic reconfiguration. The selected communication model is based on the Event Notification Pattern [10] because the smoothly integrated implicit invocation mechanisms scales well from the most basic to very large systems.

1.1 Requirements

The **HIDOORS** event service is used to provide event notification between processes on different processors. The structure for event handling, with receiver and transmitter (see chapter 4), is designed to enable communication via events over a realtime bus with low latency. Once received, events are passed quickly to special realtime threads to handle with strict timing guarantees.

The service decouples sender and receiver to provide for flexibility and simplified code reuse. To provide a easy to use communication infrastructure for different application nodes or components without direct links a logical channel definition is necessary. Different functional domain and application communication may share common network access for each node.

Network management is distributed. Each node has its own managing unit, thereby avoiding a centralized point of failure, like the CORBA [6] event manager. The receiver, transmitter, and event channel administration classes are implemented in Java and provide direct access to the communication bus through a Java API.

Each event is designed to fit in a minimal number of network package, preferable a single package, to meet realtime requirements of distributed and embedded systems. With the known latency and access time parameters of a special realtime networks, an applications can be described by their own execution values in addition with this parameters. The event channel network provides a infrastructure that allows an easy worst case execution time analysis for distributed aspects in advance, while application design phase.

1.2 Comparable Development

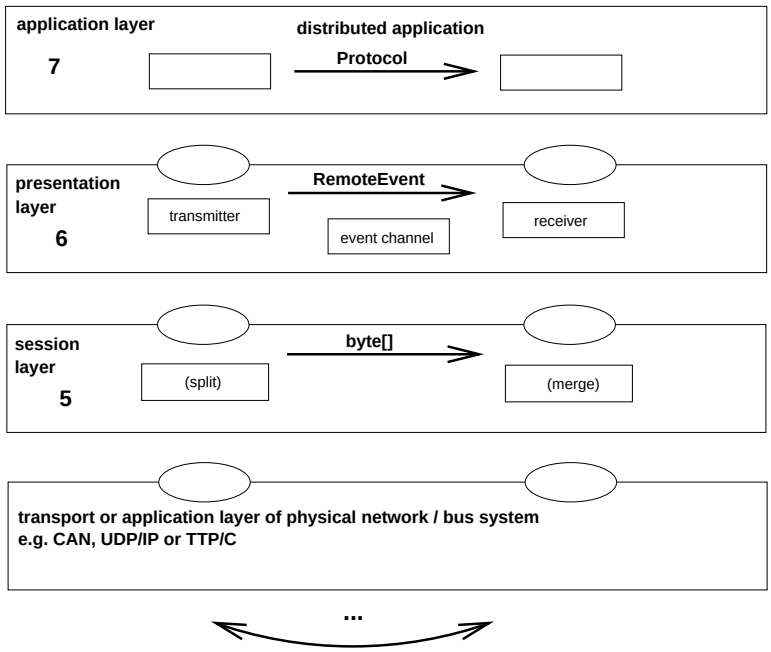
Since the design of the event service, other comparable developments in the area of event networks and implicit invocations have become available. CORBA was known at the time of conception, but a realtime variant has become available. **InfoBus** is another similar communication system that has also emerged since the **HIDOORS** project was conceived.

The Object Management Group (OMG) specified an event service model for CORBA [5]. This CORBA model is more general in that it leaves open several design choices, it accepts both typed an untyped events while the event channel network introduces static typing to event notifications over named channels.

To support developers in meeting realtime requirements by facilitating the end-to-end predictability of activities in the system and providing support for management of resources the OMG recently announces the Real-Time CORBA Specification [7]. This specification is defined as extension to CORBA 2.2 and the Messaging Specification and promises now a high level of flexibility, portability, and interoperability for realtime system development.

Another bus system known as **InfoBus** [13], used in the area of application design with JavaBeans, has a comparable design to the event channel network. It is used for the wire between several components where each component can act as data producer and/or as data consumer. By using the **InfoBus**, the cooperating JavaBeans works together without discover, introspection, and direct methods calls. The **InfoBus** provides an mechanism similar to the event channel network with events and a communication bus, but the concept of logical channels is missing. The newly available version 1.2 of this API would meet much of the needed requirements, but current implementations are also not deterministic.

Figure 1.1: event channel network application (layer structure)



The presented solutions for event notification have a lot of similarities with the event service but a broader and heavy design or other shortcomings limit their applicability to realtime monitor and control applications.

1.3 Architecture

The **HIDOORS** event service is organized in two main layers as depicted in figure 1.1[1]. The *session layer* directly interacts with the underlaying communication network infrastructure and provides an byte array access and a multicast communication with logical channels. On top of this, the *presentation layer* gives the application access and provides a logical view based on events.

The event channel network design can be illustrated in terms of the ISO/OSI layer structure (see table 1.1). In the ISO/OSI reference model, the event channel network session layer works on top of the four lower layers[2] of any communication network or bus system and provides a one to many multicast network. The presentation layer extends the session layer with control mechanisms and adds structure to the units of data that are exchanged (Java objects). It provides a service API to

[1] This figure shows the communication only in one direction.

[2] A Controller Area Network controller only provides OSI layers 1 and 2 where the event channel network is attached with an adapter.

Table 1.1: ISO/OSI integration of the event channel network

ISO/OSI layer	event channel network
7 - application	**HIDOORS** application
6 - presentation	event handling
5 - session	logical channel
4 - transport	
3 - network	
2 - data link	
1 - physical	

the application layer to implement distributed realtime object-oriented applications with Java technology.

The basis of the network design is an easy to implement socket to transmitting byte arrays over a network. A generic interface is used to ensure network independence. The bytes transmitted are used to encode events and channel membership information in the service API. A wrapper structure is used to extend the simple protocol used with the network to the full Java interface. Each system node use a single receiver and transmitter.

The implementation is generic to ensure the usability with different communication networks. The socket interface it uses, bridges from the services provided by the underlying network to the session layer of the event service. This enables the use of a wide variety of underlying networks. This is reflected in the discussion of error handling (Chapter 4) after the main components of the event service are presented. The concrete implementation of this architecture with classes, interfaces, exceptions and theirs functionality structured by packages is tabulated in appendix A.

Chapter 2

Socket Foundation

To guarantee a platform and network independent event service, the event channel network defines a socket API as the foundation of the service. Implementing this API is the minimum requirement for it usage, but realtime behavior requires the event channel network to be built on top of a (broadcast oriented) realtime bus. The API builds on top of any transport layer with packet transport features. The session layer of the event channel network provides a one to many multicast communication model with logical channels. The layer is responsible for the byte array broadcast, and the node-side grouping in logical channels.

Different nodes and components are connected with an implementation of the **`BroadcastSocket`**. This is the main interface for network access and it is modeled on **`java.net.MulticastSocket`** to receive and send byte arrays. Depending on the network architecture, the concrete implementation of **`BroadcastSocket`** handles the split and merge of successive network packages to provide a full byte array access. Features provided by the network are used to facilitate the best quality of service.

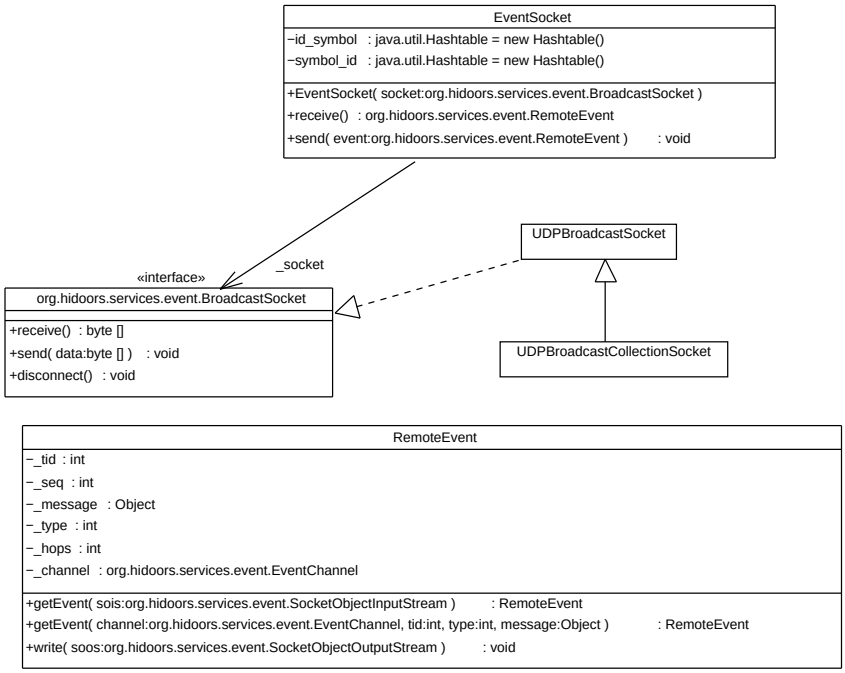
The concrete implementation of **`BroadcastSocket`** for UDP/IP [9, 8] multicast[1] is the **`UDPBroadcastSocket`**, the **`CANBroadcastSocket`** makes use of the package structure over the CAN [1] and uses successive packages to transmit a byte array. This implementation also adds missing transport layer functionality to the CAN to provide a byte array transmission with more than 8 bytes.

To extend this basis network access to support logical channels and true Java object transmission, the **`BroadcastSocket`** is encapsulated by an **`EventSocket`** to send and receive **`RemoteEvents`** objects. This wrapper guarantees the selection of events depending on logical channels. Figure 2.1 shows the class diagram for the generic interface, an implementation based on UDP/IP, and the relationship to the **`EventSocket`**.

The **`EventSocket`** forms the basis upon which logical event channels are built.

[1]Though UDP/IP is not a protocol usually associated with realtime busses, there are realtime ethernet implementation that can use it.

Figure 2.1: Sockets and RemoteEvent



Chapter 3

Logical communication channels

The event channel network is based on logical channels (**`EventChannel`**) for communicating between Java Virtual Machines. In the session layer, the channel is used to group nodes, on one or more communication networks or bus systems, into a logical network for exchanging events associated with the group and the declared channels. The session layer limits a broadcast in the transfer layer to a multicast depending on channel membership.

Figure 3.1 shows an abstract example with two different channels (channel A, B) over three communication networks (CAN, UDP/IP 1 and 2) with four nodes (node 1 to 4) connected with suitable sockets (see chapter 2). Node 3 and 4 communicate over a CAN interface and node 3 propagates the information over the channel B.

From the programmers perspective, an event channel is named by a unique String, however, for efficient event encoding, this name is mapped to an integer channel identifier, which may differ between communication networks and is only unique within a given communication network. The mapping between **`EventChannel`** identifiers and the String symbols used to identify the channel is managed by the associated **`EventSocket`**.

This mapping had to be negotiate within the creation phase of the logical channel between all participants. Since the event channel network provides a dynamic configuration infrastructure, a common communication channel for administration with a dedicated event protocol is defined. To transfer the symbol-id pair during the creation phase of a **`EventChannel`**, an externalizable Object is used. The local symbol and a randomly created identifier is packed in an **`IDSymbol`** object. For information about the channel creation protocol, see section 4.4. To cope with the restrictive capacities of embedded systems, the number of channels is limited by **`IntegerFactory.MAX_VALUE`**. The slots are filled with unique identifiers for each symbol in any given **`EventSocket`**. The restriction is due to the use of a factory to create flyweight objects to identify the channel objects in a hash table structure.

The **`EventChannel`** is used to create a logical communication channel over one or more **`EventSockets`**. It must be created with the

Figure 3.1: Abstract example with logical channels and different communication networks

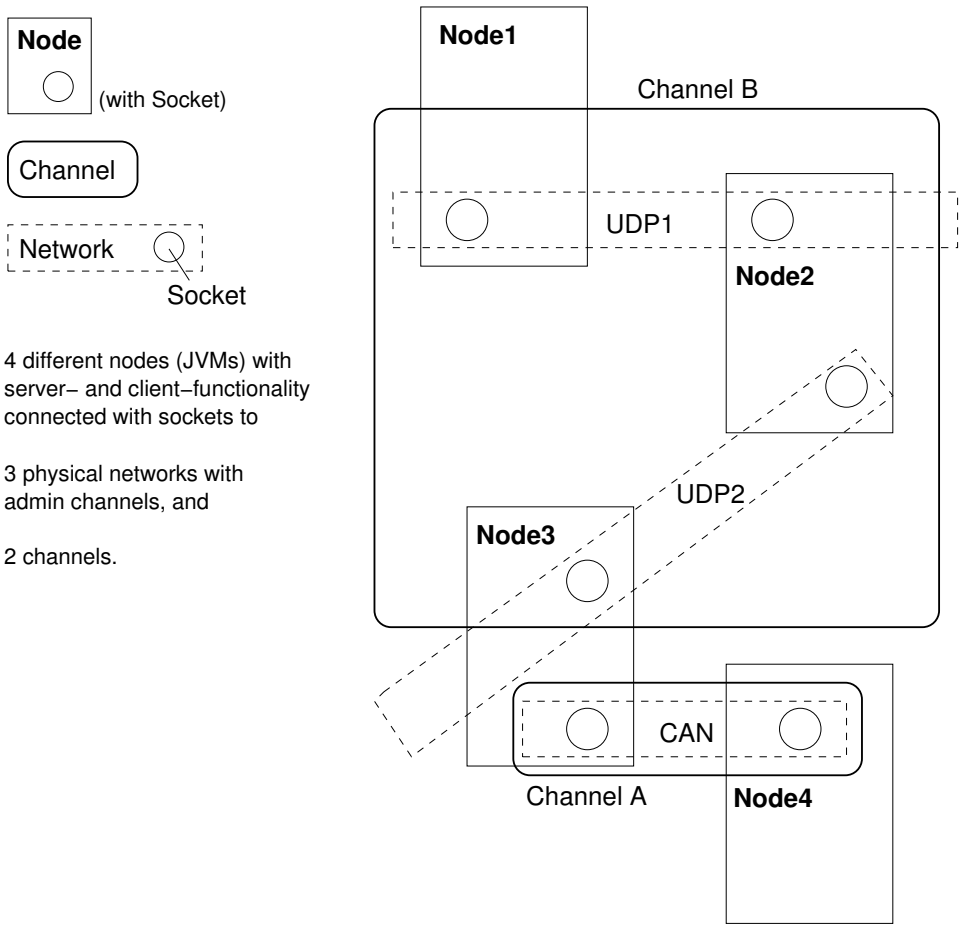
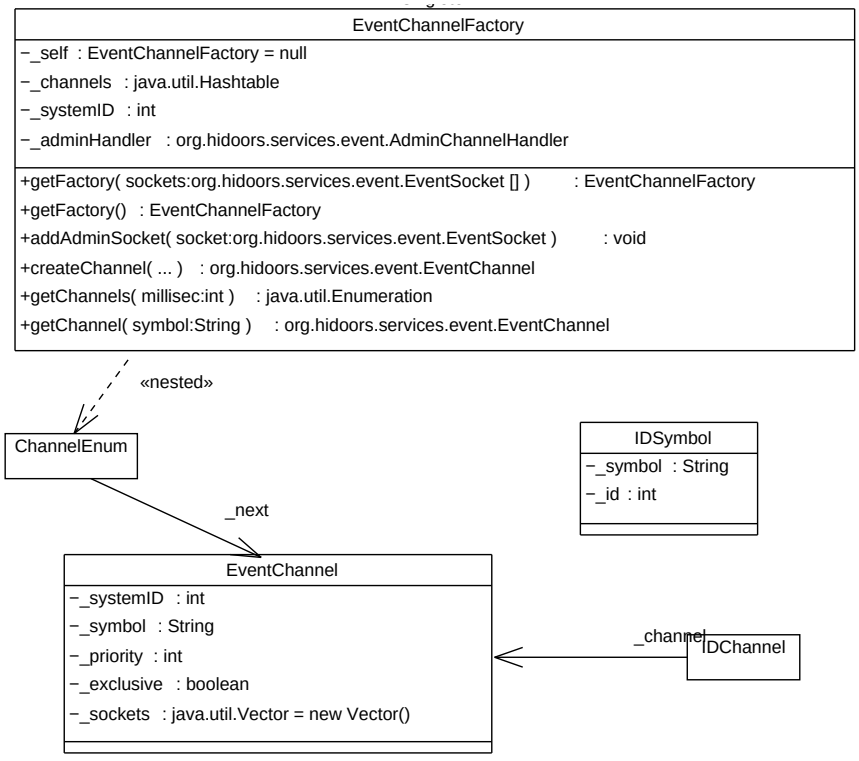


Figure 3.2: EventChannel, EventChannelFactory, IDSymbol



`EventChannelFactory`. The channel symbol is unique system wide. The interface of the classes used is shown in figure 3.2.

Each channel has the following attributes:

> **`_symbol`** for identification, e.g., for searching the local database of known channels;
> **`_priority`** for ordering the processing of the received events;
> **`_systemID`** to identify the creator node;
> **`_exclusive`**, a flag used for writing only by the creation node; and
> a transient list of local bound **`EventSockets`**.

A table of registered channels is kept in each of the **`EventChannelFactory`**, one per virtual machine. This factory is a singleton in each node.

As can be see, Java components can communicate with other application modules

and nodes using the local infrastructure with a list of known **`EventChannel`** references and the event types defined in each channel. This forms the base of passing events between network nodes, but its direct would be inconvenient. Each application should only have to concern itself with send and receiving events in its own logical channel.

Chapter 4

Event Handling

Once the channel network is established, a logical event handling system can be built on top of it. This is the presentation layer of the event service and it is responsible for providing the application with a logical view of the events it needs to send and receive. A well defined receiver, transmitter, and handler infrastructure supports the event channel assignment of events and handling actions in the each node of the network. Applications need only concern themselves with sending their own events and registering handlers for events in which they have an interest.

4.1 Events

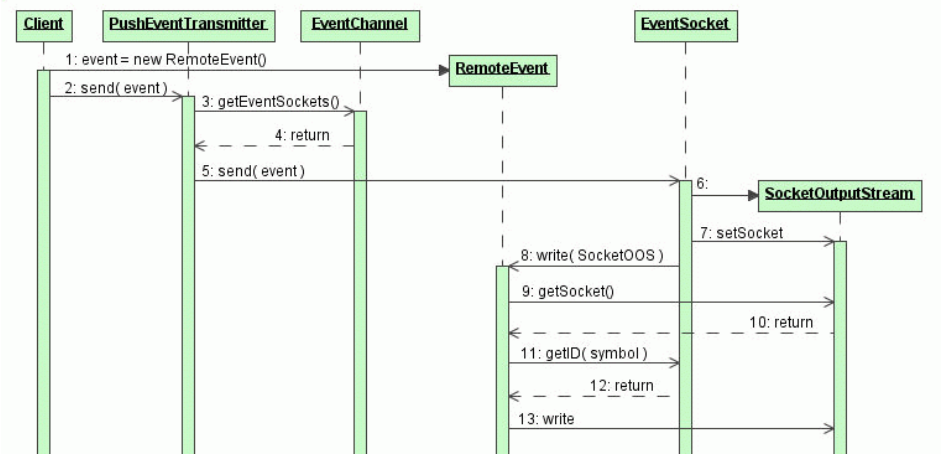
The basis of the event service is the event. The Java objects transmitted over the event channel networks are **`RemoteEvent`** objects. This class **`RemoteEvent`** provides a factory method **`getEvent()`** for creating objects. This method initializes the new object with a reference to the channel and the message to be send.

The serializable **`RemoteEvent`** has the following attributes (with accessor methods):

> *identifier* created with *sequence number* (**`_seq`**) and *system id* (**`_tid`**),
> *hop count* (**`_hops`**) indicating hops in router nodes between different communication networks,
> **`_type`** associated with the channel and known in the handler class,
> **`_message`** (serializable *Object* to send), and
> reference to **`EventChannel`**. The reference to the **`EventChannel`** object is not serialized for transmission, because this reference would be invalid in any other memory space.

The main goal of the lightweight event service protocol is a network independent communication with low latency and the communication overhead kept to a minimum. For this reason, the serialization of the **`RemoteEvent`** objects is not managed via the Java **`java.io.Serializable`** interface. The instance method

Figure 4.1: RemoteEvent Serialization



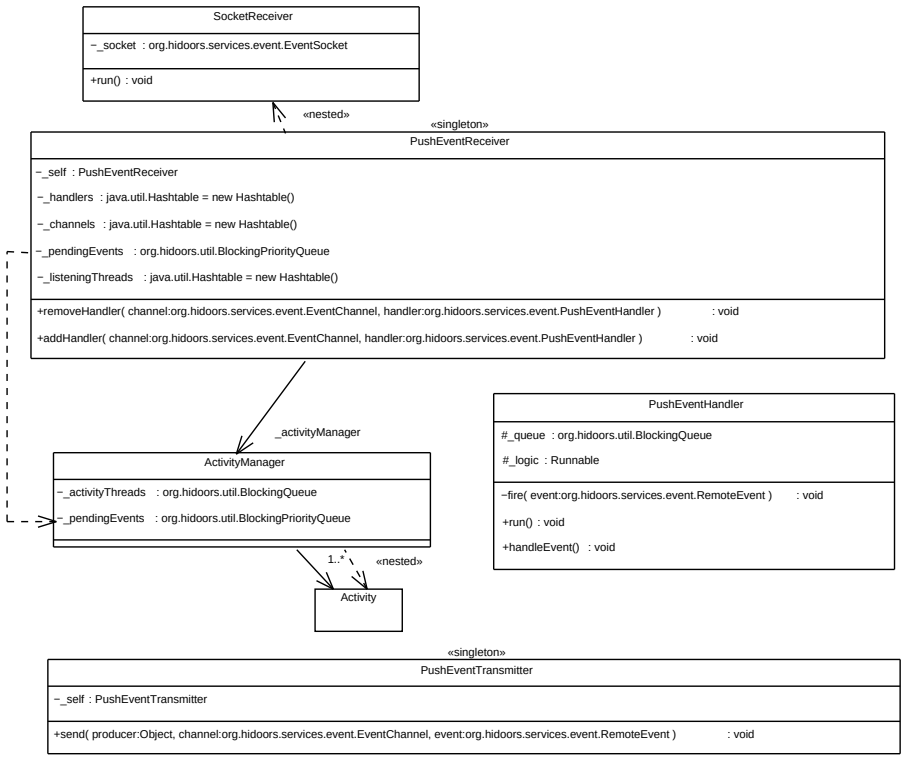
`write()` writes an event to a **`SocketObjectOutputStream`** and the factory method **`getEvent()`** is used to reconstruct a serialized event instead. The Java object is encoded using predefined integers. The figure 4.1 shows the application flow of the serialization.

4.2 Handling Infrastructure and Process

To support event channel association and assignment to special handling code, the presentation layer of the event channel network extends the socket communication mechanism with additional administration components. This infrastructure is designed to enable swift event consumption. Each event is passed without delay to an executable class object to be handled.

Each node, both server and client, has one **`PushEventReceiver`** to listen for events and one **`PushEventTransmitter`** to transmit events on each registered **`EventSocket`** via its encapsulated **`BroadcastSocket`**. The handling of incoming events is done with an instance of **`PushEventHandler`** or a specialization thereof (see figure 4.2). When an event is received, the handler's **`fire()`** method is called to add the event to an internal **`BlockingQueue`**, then the handler is started in another thread where the **`run()`** methods is called, which calls its **`handleEvent()`** method in turn. The **`handleEvent()`** method invokes the handlers **`Runnable`** object. Since **`run()`** is final, this behavior can only be changed by overwriting the **`handleEvent()`** method in a subclass of **`PushEventHandler`**. The **`PushEventHandler`** is modeled on along the lines of the thread model of the realtime Java specification. A separate thread is used for each channel priority to handle the events received.

Figure 4.2: PushEventHandler



The **PushEventTransmitter** implements methods for sending events to the socket. It is used by a user program to take care of message delivery. The framework can also use the transmitter itself. For example, when the **EventChannelFactory** is called to create a new exclusive **EventChannel**, a channel create request must be sent (compare section 4.4).

On the client side, the **PushEventReceiver** starts a listening thread on each registered **EventSocket** via **PushEventReceiver.SocketReceiver** to receive the incoming **RemoteEvents**.

An **EventSocket** is processed only once a **EventChannel** is created bound to the socket. The **PushEventReceiver** binds at least one handler to each known socket for the administration channel when the physical infrastructure is initialized for the local node. The **EventChannelFactory** provides a special method for this purpose: **addAdminSocket(EventSocket)**. The action is then passed to the **PushEventReceiver** and a new receiver thread is started.

For low latency, the event reception is a two stage process. First, the **PushEventReceiver** reads each **RemoteEvent**. A **RemoteEvent**, for which a handler has been registered, is put in a **Queue** for the **ActivityManager** to dispatch. Then the **ActivityManger**, another thread running on the node, initialize a waiting realtime thread out of a pool with the selected **PushEventHandler** and starts it to handle the event. For each **PushEventHandler** registered for the event being dispatched a waiting realtime thread is used. The number of these waiting **ActivityManager.Activity** threads is set with the constructor of the enclosing **ActivityManager**.

The **ActivityManger** is a thread that has access to the blocking priority based queue of pending events and a pool of waiting threads. If the **PushEventReceiver** puts a received RemoteEvent in the queue, the **ActivityManager** starts a waiting thread with the appropriate handler. It pops the event out of the pending event queue and puts it into the queue of the **PushEventHandler**. The **PushEventHandler** can then handle each event in turn.

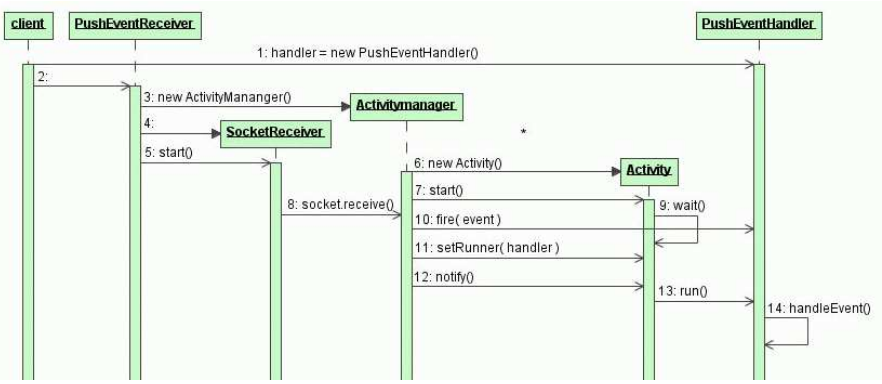
The queue of **RemoteEvents** between the **PushEventReceiver** and the corresponding **ActivityManager** is priority ordered, i.e., sorted by the priority of the associated channel.

The sequence diagram in figure 4.3 shows this application flow for deserialization during the reception of an event.

4.3 Queues

The event channel network makes intensive use of queuing Java objects. For this purpose, several data structure classes are included as part of the distribution (see section 4.3). Several queue classes are defined to support managing asynchronous

Figure 4.3: Sequence Diagram for Remote Event Reception



communication. They are all based on a generalized **`Queue`** class. Every queue has a **`put()`** and a **`pop()`** method to insert and remove elements. With **`get()`**, the next element to pop can be retrieved without disturbing the queue.

`BlockingQueue` specialize **`Queue`** and implements blocked reading. The **`pop()`** method waits until at least one object is put in the queue.

`BlockingPriorityQueue` contains a array of **`Queues`** and provides a blocking read with different priorities. The priority is defined with an Integer object (a flyweight also created with the **`IntegerFactory`**). The maximum identifier is the highest priority. The first object in the associated queue is popped before those in lower priority queues.

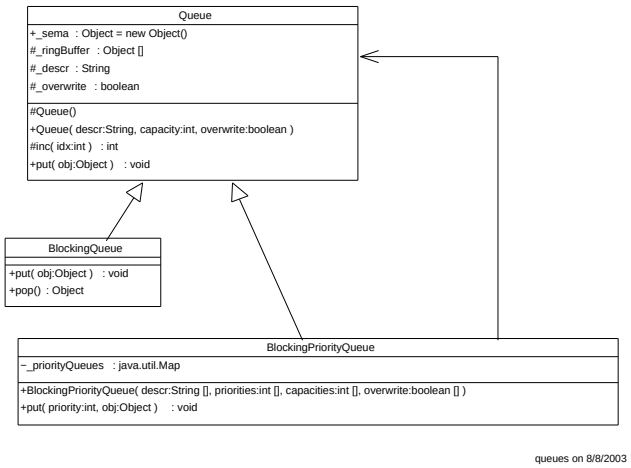
The figure 4.4 shows this classes and there relationship.

4.4 Administration Channel

The decentralized structure of the administration in the event channel network and it's applications, localized in each virtual machine, guarantees no central point of failure but it also requires a mechanism for the creation of a global view of information. For this purpose, the system maintains a special communication channel with an integer identifier of 0 for coordinating the administration of the event server components between nodes.

The system has a special handler for the administration channel which is registered when the first call to the **`EventChannelFactory`** is made. In the interface **`AdminChannel`**, type information for the events and the properties associated with this channel are collected. **`AdminChannelHandler`** implements the **`PushEventHandler`** for these events and provides convenience methods for sending necessary events via **`PushEventTransmitter`**.

Figure 4.4: Queue



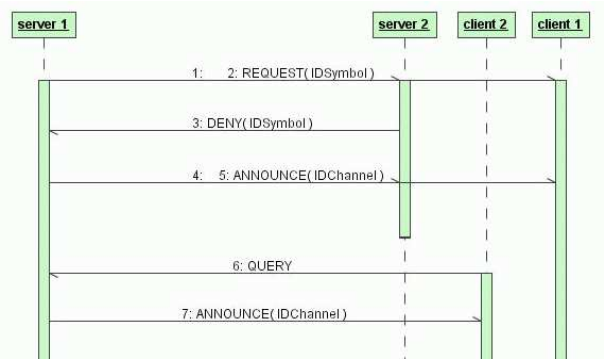
The main task of the administration channel is the creation of new channels. The symbolic name of a new channel must be unique system wide and each **`BroadcastSocket`** needs a unique id for the encoding and communicating over its communication network or bus system. Also the administration event types must be know head of time. These types for the **`RemoteEvents`** associated with this channel are the following:

1. normal MESSAGE,
2. QUERY for announces,
3. ANNOUNCE of a **`EventChannel`**,
4. REQUEST to create a new **`EventChannel`**, and
5. DENY the creation, because channel exists and is in exclusive use.

`AdminChannelHandler` is simply a **`PushEventHandler`**, which is registered on the local receiver for each used **`EventSocket`**.

New channel creation uses a protocol defined on the administration channel to establish unique a reference and deliver the channel's configuration to all nodes. The **`EventSocket`** is also used as repository for the identifier to symbol mapping. The figure 4.5 shows the communication flow between four nodes to create a new channel.

Figure 4.5: Administration Channel Protocol



If a server 1 wants to create a new channel, the **`EventChannelFactory`** sends a REQUEST event with the symbol String and a new integer identifier wrapped in a **`IDSymbol`** object over each **`EventSocket`** registered with the local administration channel.

All connected peers (server 2, client 1, client 2) receive the REQUEST event via their **`PushEventReceiver`** singleton. If another server 2 has already registered another symbol with the same identifier value on the socket, it can send a DENY event to reject the creation. If server 1 does not receive any DENY event in a specified time slot, it sends an ANNOUNCE event with a **`IDChannel`** object. The **`IDChannel`** itself is a wrapper for the accepted identifier and a **`EventChannel`** object serialized without a reference to a local **`EventSocket`** object.

`EventSocket` objects have to be added at each node when an ANNOUNCE event is received. If client 2 comes up later, it will not receive this events. It must send a QUERY event to solicit ANNOUNCES form all other nodes before it tries to initialize its own channels. All servers with active channels must respond to the QUERY with ANNOUNCE events.

4.5 Miscellaneous Utility Classes

To complete the view of classes in the library necessary for the presentation and session layer of the event channel network, there are also some utility classes to provide support for configuring. **`IntegerFactory`** is a simple factory class to create flyweight Integer objects used with the symbol identifier mapping in **`EventSocket`**. **`CodeGeneration`** is used with the XSL transformation (conceptually described in chapter 5) and provides an infrastructure to create and compile temporary Java files.

Chapter 5

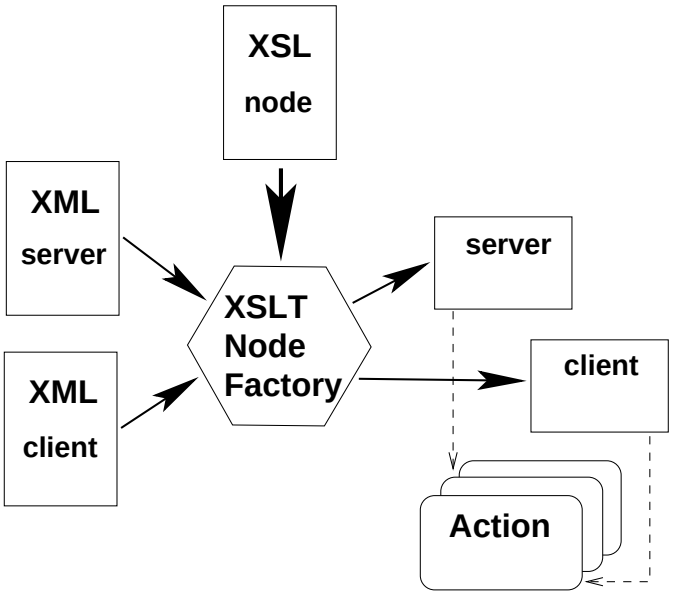
XSL Transformation

In order to provide for dynamic application development, the event channel network supports an XSLT based software description and generation model. With the classes introduced in this chapter, it is possible to generate applications (servers and clients) with XSL transformations from XML descriptions and additional user provided action classes.

A formal description of the XSL transformations to generate application software for the different nodes is defined in a DTD given in appendix B.2. With additional action classes, specializations of the **`AbstractAction`** factory class, the programming of a node software with server and client components could be simplified. The figure 5.1 shows the involved components and the process with a XSL transformation. This API is a first step to the later code generation from UML diagrams in the **HIDOORS** profile.

`XSLTransformationFactory` is the base class for XSL transformation based factories to create Java applications from describing XML files. **`XSLTNodeFactory`** in conjunction with the XSLT file **`node.xslt`** enables the transformation of XML files to simple applications using the event channel network. The abstract class **`AbstractAction`** provides an standardized way for creation and initialization objects and is used as generalization for actions referenced in the XML description. For an example, see section 7.3.

Figure 5.1: XSLT components and process



Chapter 6

Failure Modes and Error Handling

For the purpose of analyzing its error handling and failure modes, the event channel network can be viewed in terms of the ISO/OSI reference model for computer communication. The table 6.1 shows the event service's functional classification in layer 5 and 6. It relies on the underlying layers for network protocol with transport layer functionality. Error handling benefits from this layered structure and transparent lower level error handling. Error handling in the event service is modeled for three different networks (as depicted in table 6.1) with a discussion of dependent errors and recommended error handling.

For the UDP/IP network, the broadcast socket implementation of the event channel network builds on top of the transport layer with an unreliable package delivery protocol. Delivery and duplicate protection have to be done in the event service and the application. In order to minimize the problem with different delivery routes and the resulting duplicate packages, an event is packed into a single network package. While the UDP/IP protocol enables complex network topologies with several subnetworks and different routes for network packages, the network topology used with distributed applications in the **HIDOORS** project is rather simple with no need for routing.

CAN is a simpler protocol than UDP/IP. Though there are some extensions to CAN, only layers 1, 2 and 7 are specified in the base protocol. Since layer 7 is the application layer, only layers 1 and 2 are used with the event service. To cope with the maximum of 8 byte user data packets in a CAN data frame, an event service adapter is used with a split/merge functionality to send larger events. This functionality is integrated in the appropriate broadcast socket (`CANBroadcastSocket`).

The Time Triggered Protocol, TTP/C, is the only realtime protocol considered. The event channel network is integrated on top of the Protocol Service Layer of TTP/C. That layer provides a packet oriented, fault tolerant, realtime communication service [15]. The maximum user data size in network packages of 236 byte enables the packing of an event in a single frame. Since the event-driven design of the

Table 6.1: ISO/OSI reference model vs. event channel network with UDP or CAN

ISO/OSI Layer	event channel network / UDP	event channel network / CAN	event channel network / TTP/C
7 - application	**HIDOORS** application		
6 - presentation 5 - session	event handling logical channels		
4 - transport 3 - network	UDP IP	event service adapter	Protocol Service Layer
2 - data link 1 - physical	Ethernet	CAN	Data Link Layer Physical Layer

event channel network does not correspond completely to the time-devision nature of TTP directly, the implementation for the broadcast socket interface is used to adapt this access schema similar to[4].

The event channel network itself provides no active error handling. All basic modes of failure for messages (depicted in figure 6.1) are forwarded to the application layer. Failures in the session and the presentation layer are treated in the same way.

There are three broad classes of systemic errors: faulty transport, faulty device, and interference (e.g., crosstalk). Though a system can not necessarily distinguish between these classes, the set of secondary errors can be detected by the system.

1. A packet is lost resulting in a sender/receiver time out on transmission or reception.

2. The underlaying network reports an access failure, e.g., the node physically fails.

3. The presentation layer reports protocol failure like unknown event channels or event types.

4. A message could be too large to send.

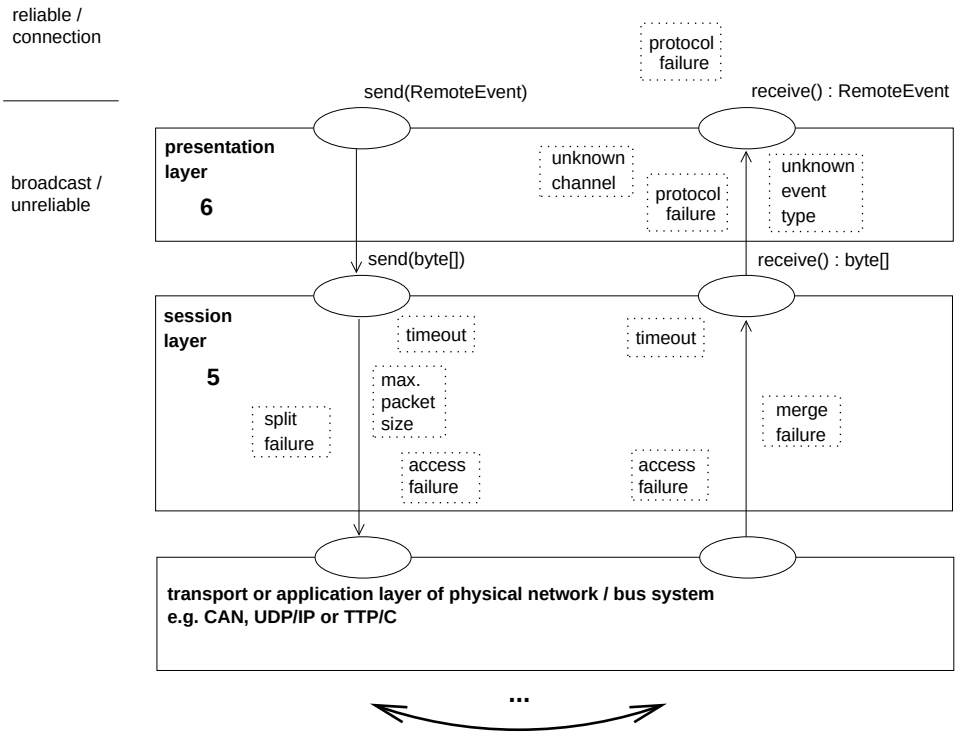
5. The failure could occur while splitting or merging the data, e.g., the receiver gets a mangled packet.

6. The transmission fails, e.g., a packet has been dropped.

7. The protocol is abused, e.g., a babbling idiot resends a packet again and again.

8. A Node acts faulty, e.g., always send, never send.

Figure 6.1: Socket communication and possible errors



The first five cases can be detected by the event service or the underlying network and an exception can be thrown; however a number of failures in the last three cases could be undetectable. If a whole message (Java object) is missing, this must be detected at the application level. Indeed, to support decoupling of sender and receiver, the application must be aware of event frequency and react to missing messages accordingly. A protocol failure like a babbling idiot could only be detected through the sequence number in a transmitted event within the receiver or transmitter or at least at application level.

Detecting these transmission failures in the event channel network is based on timeouts and the RTSJ high-resolution time API. The **`write()`** and **`receive()`** methods provided in the **`BroadcastSocket`** interface and the classes implementing it for different communication networks guarantee the timing limits of the underlaying communication network infrastructure and throw an **`IOException`** for any anomaly. The driver class implementation makes use of features of the underlying communication system. For example, **`UDPBroadcastSocket`** boxes each message in one network package to avoid the merging of missordered packages. The CAN implementation uses the CAN message ID to denote a whole **`RemoteEvent`**.

Exceptions must be handled in the application layer. The application must know how frequently messages should arrive, and take appropriate action when they do not. The arrival rate should be well within the safety time for the system to ensure that corrective action can be taken in the event of a failure in the underlying network.

No high level error handling is integrated in the event channel network. The application provides **`PushEventHandler`** code and the infrastructure uses the **`EventSocket`** with an encapsulated **`BroadcastSocket`** implementation to pass events and exceptions to the application itself. Here, a failure that violates any application level assertions has to be handled. Figure 6.1 shows the layered structure of the event channel network and a diagram with actions and possible error points for a socket communication.

Conceptually there are two cases of events and necessary error handling mechanisms that can be used with the event channel network.

1. For regular events, sent frequently with a well defined period, the receiving node must detect and react when messages do not arrive within that period. If an expected event is missed, it could be lost or the sending node could be down. The receiving node application must react accordingly. A use case for a connection oriented application with error handling is shown in **`org.hidoors.test.timeout`** test scenario (see section 7.1.4). This server is controlled by a client and if a timeout is reached and the client has received no further events, it tries to restart/reset the server with an event.

2. For events irregular frequency, the receiver has no way of knowing when an event should arrive, therefore only the server can notice errors. For this

to work, each event must be acknowledged by the receiver and the receiver must know how many acknowledgments it should receive. This mimics the case of a standard RMI message except that the sender and receiver do not have to have explicit knowledge of one another.

The latter mode is more difficult to implement correctly and should be used more sparingly.

As with th IP Internet protocol, the event channel network provides an unreliable packet delivery service. Any failure repair of missing or missordered messages has to be implemented in higher application levels. Nevertheless, for existing reliability and connection information in a physical realtime network, a specialized **`BroadcastSocket`** with a cooperating **`EventSocket`** could be designed. This would enable the packaging and analysis of additional control information in transmitted messages.

By using the ISO/OSI layer reference model [11], one can illustrated the kinds of errors that can occur and how they are be handled on the underlaying network protocol. Here UDP/IP, CAN, and TTP/C will be considered for the first four layers, since the event service builds on top of their respective transport layer functionality. Possible errors in the session layer and the presentation layer are discussed once for all protocols since the error recovery in the event channel network layers is the same regardless of the underlying protocols. The application layer must handle all remaining cases.

6.1 UDP/IP over Ethernet

Though UDP/IP is not a network protocol usually associated with realtime busses, there are projects to extend Ethernet with realtime characteristics [16] which could host UDP/IP. Standard Ethernet provides multiple access with carrier-sense multiple access and collision detection (CSMA/CD) to avoid conflicting media access of different nodes [12]. Because of the cost effectiveness of Ethernet, many improvements have been made to enable Ethernet to support time constrained communication. This implementation of the event channel network using UDP/IP works on any UDP/IP and Ethernet implementation. Of course, with standard Ethernet, there are no deterministic realtime guarantees. UDP/IP itself does not guarantee a reliable transport layer service, but built on the connectionless package protocol with low protocol overhead, the event channel network could be tested easily. The examples in chapter 7 show its practicability, and illustrate different aspects of the functionality provided by event service. The error handling in the event service over UDP/IP and Ethernet can be illustrated by examining each ISO/OSI communication layer in turn.

6.1.1 Physical Layer

The physical layer is responsible for the bit transmission. Possible errors in this layer concern bit errors. Bits can be lost or modified as follows:

a bit is lost,
a bit is modified,
near simultaneous access can cause a collision resulting in bit corruption,
packets can be sent continuously resulting in duplication, or
the physical connection can be interrupted resulting in data loss.

Error Detection Capability of the Layer

This layer only detect when the bit being transfered does not match the bit on the wire.

Error Diagnostic Capability of the Layer

A mismatch between the bit being sent and the bit currently on the wire is interpreted as a collision.

Error Correction Capability of the Layer

When a collision occurs, the sender waits a variable interval and then sends the packet again.

Error handling Information received from upper Layer

The necessary time intervals for backoff are provided by the upper layers.

Error Information supplied to upper Layer

Only error detection information embedded in the packet are passed to the next layer up.

Undetected Errors or not handled by the Layer

The physical layer provides no integrated error detection or correction capability for errors beyond the bit transmission. If a receiving node does not get an error notification, the received bit is assumed to be correct. For a transmitting node, the breakdown of the physical network itself is not detectable at this layer. An upper layer has to detect this since no other communication partner sends an acknowledge. If a node is faulty, e.g., continuous sending, this error results only in other anomalies.

6.1.2 Data Link Layer

The physical layer has to ensure correct bit transmission as the foundation for data package transmission in the data link layer. This layer handles multiple accesses to one communication media. It serializes the packages or frames sent from different nodes. With a Cyclic Redundancy Check (CRC), the correctness of the transmission is tested. Error at this level are as follows:

an error passed from the physical layer,
a packet was mangled, or
an extraneous network package is received.

Error Detection Capability of the Layer

Error detection only results from the bit monitoring error notification of the physical layer and from a packet transmission control. Each example network provides a package structure and protocol at this layer. It handles an incorrect transmission via an error protocol with retransmission or timeout. If the package structure is mangled, the receiving node detects this with checksums and drops the package.

Error Diagnostic Capability of the Layer

No diagnostic capabilities exist in this layer.

Error Correction Capability of the Layer

There are modes for sending an acknowledgment for correct packets at this level. Though some realtime versions of ethernet take advantage of this facility, acknowledgments are normally not used with IP. Without packet acknowledgment, a sending node can not initiate a retransmission and hence no error correction is provided.

Error handling Information received from upper Layer

The necessary timing parameters are provided by the upper layers.

Error Information supplied to upper Layer

When faulty packets are received, the next layer up is notified.

Undetected Errors or not handled by the Layer

This layer can only detect errors resulting in mangled packets. Missing packets and duplicated packets are not detected and the handling for these errors has to be done in higher layers. Also, packets which pass the CRC could still be damaged. Upper layers have to detect these errors as well.

6.1.3 Network Layer

In a UDP/IP network, the network layer is responsible for end-to-end data transfer across all intermediate networks. It is responsible for breaking large packets into sendable fragments and reassembling fragments into full packets. Only duplicate and miss packet fragments can be handled at this level.

Error Detection Capability of the Layer

Errors detected in this layer concern problems with end-to-end communication in networks organized with connected subnetworks. Because the event channel network depends on small networks of embedded systems and controllers this errors can be detected with IP but they are implausible and not relevant. Only the detection of duplicate packet fragments is relevant.

Error Diagnostic Capability of the Layer

Diagnostic capability is confined to the reassembly of packet fragments into full packets.

Error Correction Capability of the Layer

The errors concern missing and duplicate network package fragments. Duplicate fragments and packets are ignored. IP uses ICMP (Internet Control Message Protocol) to notify the sender when any other delivery errors including errors flagged from an underlying layer. The sender then resends the packet.

Error handling Information received from upper Layer

This layer guarantees end-to-end connections in known time limits. The necessary time restrictions are provided by the upper layers. If the access (receive/send) failed because of timeouts this error is reported to the upper layer.

Error Information supplied to upper Layer

Problems concerning routing of network packages resulting in a timeout are conferred to the next layer up.

Undetected Errors or not handled by the Layer

IP provides checksum control for only the IP header. Further control have to be done in transport layer.

6.1.4 Transport Layer

UDP acts as transport layer and provides a connectionless service for application-level procedures. UDP is basically an unreliable service which does not guarantee delivery. Due to the low protocol overhead of UDP, the error detection capabilities are limited to a simple checksum algorithm. If any error is detected, the segment is discarded and no further action is taken.

Error Detection Capability of the Layer

The error detection in UDP is based on the same checksum algorithm as in IP, but for the whole package. Mangled packages are simply discard, with no notification for the upper layer.

Error Diagnostic Capability of the Layer

There is no further diagnostic capability.

Error Correction Capability of the Layer

There is no correction capability integrated in this layer. UDP does not request the retransmission for a mangled package. Lost packages are not recognized.

Error handling Information received from upper Layer

Informations received from the upper layer again refer to timing constraints.

Error Information supplied to upper Layer

UDP provides a connectionless service and detected errors are not handed on to the upper layer. If an error is detected, UDP discards the affected network package, without any notification to the upper layer. Only when a transmission times out is the next layer up notified.

Undetected Errors or not handled by the Layer

UDP provides no guarantee on package delivery, therefore lost packages must be detect in the upper layers.

6.2 CAN

The Controller Area Network (CAN) [1] is a serial communications protocol for sending and receiving short realtime control messages. CAN is a multimaster

broadcast bus where a number of processors are connected to the bus. For multiple access, a mechanism with carrier-sense multiple access, and collision detection with arbitration on message priority (CSMA/CD+AMP) is used. Since the design of CAN is limited to ISO/OSI layers 1 and 2, the event channel network is connected through an event service adapter which is integrated in the broadcast socket implementation for CAN. The class **`CANBroadcastSocket`** implements the generic **`BroadcastSocket`** interface and adds the missing functionality to transmit larger byte arrays over the protocol. In this way, the socket design provides an uniform view on the underlying network layers. The CAN implementation provides for the same interface as for other underlying networks, e.g., the UDP/IP implementation.

6.2.1 Physical Layer

The CAN communication protocol is designed to broadcast short messages (between 1 and 8 byte user data) periodically, sporadically, or on demand. The error classes here are similar to those of ethernet. Bits can be lost or modified as follows:

a bit is lost,
a bit is modified,
a collision can result in bit corruption,
packets can be sent continuously resulting in duplication, or
the physical connection can be interrupted resulting in data loss.

Error Detection Capability of the Layer

Each controller uses bit monitoring to detect bit errors during its own transmission. It uses two mechanisms for error detection at the bit level: monitoring and bit stuffing.

Monitoring is used by the transmitter to detect errors in bus signals. When a station transmits, it observes the bus level and can thus detect differences between the bit sent and the bit received. This permits reliable detection of global errors and errors local to the transmitter.

Bit stuffing is used to insure that the bus changes state periodically regardless of the data being sent. CAN uses a "Non Return to Zero (NRZ)" coding. However state transitions are needed to insure synchronization. These synchronization edges are generated by means of bit stuffing. When a consecutive sequence of five equal bits is transmitter, the hardware inserts a stuff bit with the complementary value into the bit stream. The stuff bit is then removed by the receivers.

If one or more errors are discovered by at least one station using the above mechanisms, the current transmission is aborted by sending an "error flag". This prevents other stations accepting the message and thus ensures the consistency of data throughout the network. After transmission of an erroneous message that has been

aborted, the sender automatically attempts to retransmit the message. There may again competition for bus allocation.

Error Diagnostic Capability of the Layer

Though bit monitoring enables the detection of bit errors due to contention or other transmission errors, no further attempt is made to diagnose errors.

Error Correction Capability of the Layer

If a bit error is detected while transmission of the user data, the sending CAN controller stops transmission, sends a error protocol to notify the communication partners, and tries a retransmission later. If a receiving node gets no error signals, the controller treats the received bits as correct.

Error handling Information received from upper Layer

The necessary time restrictions are provided by the upper layers.

Error Information supplied to upper Layer

The CAN controller tries to retransmit data when the serial media is idle. If the retransmission fails and a time limit is reached, the upper layer is notified.

Undetected Errors or not handled by the Layer

The CAN protocol provides only limited error detection and correction functionality for receiving nodes on this layer. The transmitting node has to monitor its transmission and recognize bit errors. A breakdown of the physical, serial connection often result in the division of the bus, this error has to be detect on a upper layer with application protocol knowledge.

6.2.2 Data Link Layer

The data link layer is the highest layer in CAN. Usually an application communicates directly with this layer. The data link layer is responsible for transmission, framing, and error control over a single communication link. In CAN, this layer, in combination with directives to select optimal identifier orderings, guarantees correct package transmission within predefined time limits [14]. Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node, or of any intended receiving node. Instead, the content of the message is labeled by an identifier that is unique throughout the network. All other nodes on the network receive the message and each performs an acceptance test

on the identifier to determine whether or not the message, and thus its content, is relevant to that particular node.

Error Detection Capability of the Layer

At this layer, data corruption errors can be detected. Unlike other bus systems, the CAN protocol does not use acknowledgment messages but instead signals the errors immediately at then end of the packet. For error detection the CAN protocol implements three mechanisms at the data link level level: cyclic redundancy codes (CRC), frame checking, and error acknowledgment.

A CRC safeguards the information in the frame by adding redundant check bits at the transmission end of the message. The receiver recomputes these bits and tests the newly generated bits against the received bits. If they do not agree, there is an error in the data or header information.

A frame check is used to verify the structure of the transmitted frame by checking the bit fields against the fixed format and the frame size. Errors detected by frame checks are designated "format errors".

The acknowledge field (ACK) consists of an ACK slot and an ACK delimiter. The bit in the ACK slot is sent as a recessive bit and is overwritten as a dominant bit by those receivers which have at this time received the data correctly. Correct messages are acknowledged by the receivers regardless of the result of the acceptance test. If no acknowledgment is received by the transmitter of the message, an ACK error is indicated.

Error Diagnostic Capability of the Layer

When a positive acknowledgment is not received, the sender assumes that the package data is corrupt. No distinction is made between a CRC error and a framing error. Both are signaled at by at least one node holding down the ACK bit. Since the bus behaves as a wired-or, no other node can provide a positive acknowledgment.

Error Correction Capability of the Layer

The missing positive acknowledge, or a error-frame results in a automatic retransmission.

Error handling Information received from upper Layer

The necessary time restrictions are provided by the upper layers.

Error Information supplied to upper Layer

If a positive acknowledge is not received because of other communication failures, the data link layer stops retransmission after a time period, and returns with an error notification to the upper layer.

Undetected Errors or not handled by the Layer

This layer can only detect errors resulting from mangled frames, when the communication bus is up an running. Missing frames are not detected and the handling for this errors has to be done in higher layers. Also, frames which have pass the CRC could still contain damaged data. Upper layers have to detect these errors.

6.2.3 Network Layer

For the serial bus structure of simple Controller Area Networks the routing functionality of this layer is not supported.

6.2.4 Transport Layer

Since the transport layer is absent in CAN, the splitting and merging of large data packages must be done in the event channel network layers itself. This functionality of an event service adapter is implemented in the **`CANBroadcastSocket`**. Errors here concern missing or duplicate packet fragments.

Error Detection Capability of the Layer

The only functionality implemented on this layer is the splitting and merging of data packages. The **`CANBroadcastSocket`** provides a protocol that is used to pack the data parts in successive data-frames with identical identifiers. The information for merging are part of this data. The algorithm allows the detection of missing parts.

Error Diagnostic Capability of the Layer

There is no further diagnostic capability integrated.

Error Correction Capability of the Layer

If a data-frame is missing the algorithm in the socket throws an exception.

Error handling Information received from upper Layer

The necessary time restrictions are provided by the upper layers.

Error Information supplied to upper Layer

There is no error correction functionality implemented in the event channel adapter. Therefore no error information is passed up the protocol stack. The application layer must detect all errors that do not result in a retry itself.

Undetected Errors or not handled by the Layer

Since the CAN broadcast socket implements a split and merge algorithm to transmit the event in a minimum number of network packages, it can detect missing parts of an event. If the whole event (with all parts) is lost, this layer provides no further detection capability.

6.3 TTP/C

TTP/C [15], the Time-Triggered Protocol designed to satisfy SAE (Society of Automotive Engineers) class C requirements for hard realtime, is a fault-tolerant communication protocol for the implementation of hard realtime applications, and interconnection of electronic modules of distributed realtime systems. The system consists of a set of nodes interconnected by a time-devision multiple access (TDMA) based realtime communication network, with two channels. In a TDMA protocol each node is permitted to periodically use the full transmission capacity of the bus for some fixed amount of time. In its assigned time slot, a node sends frames on both channels - the frames on channel 0 and channel 1 do not have to be the same and may differ in their length and contents. The fault-tolerance strategy of TTP/C based on the so-called 'single fault' hypothesis, that requires, that any single fault – up to arbitrary loss of complete node – is reliably detected, and tolerated.

6.3.1 Physical Layer

TTP/C does not specify bit encoding or the physical media used, but certain constraints must be fulfilled.

> The TTP/C bus must consist of two independent physical channels, which may be based on different physical layers.
> TTP/C needs a shared broadcast medium.
> The boundaries of the propagation delay must be known.

Most systems in use today have high-speed CAN transceivers as underlaying physical layer. Therefore the discussion of the possible error classes in the physical layer of TTP/C is omitted.

6.3.2 Data Link Layer

The Data Link Layer deals with the access to the communication channels and the transmission of frames. This layer defines a a set of frame formats and a protocol to guarantee a global view of system state (message descriptor list, MEDL) for all participants. For correct frame transmission, a Cyclic Redundancy Check (CRC) is used. Possible error classes at this level are as follows:

the physical connection is interrupted,
a frame is mangled,
a frame is lost, or
an error occurs in the global system state.

Error Detection Capability of the Layer

TTP/C is a time-triggered protocol, and the controllers have a global system time and a well known time table for the media access. Each controller provides a comprehensive error detection mechanism for received frames. As opposed to the CAN event-triggered model, in TTP/C, error detection is not at sender but at receiver side.

Error Diagnostic Capability of the Layer

The CRC enables each controller to recognize mangled frames. A correct frame is a valid frame which passed the CRC and all additional semantic checks at the receiver. Since the CRC is calculated over the message and the current state information of the node, a correct CRC demonstrates not only that the data is correct, but also that the controller state of sender and receiver fully agree. A correct CRC indicates, that sender and receiver have been connected correctly, since otherwise the different control state values would result in CRC errors.

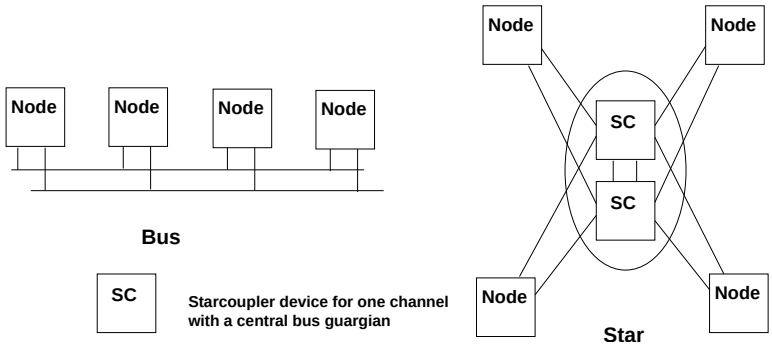
Error Correction Capability of the Layer

Each node sends status information during its transmission slot with its packet indicating the error state of the last packet received from each of the other nodes. Since at least two physical links are used, the status returned is the better of the status of the two individual channels. This information can be used by the sending node to cause a retransmission of the data or to shut itself down.

Error handling Information received from upper Layer

The correct error detection and handling information depends on the correct controller state and system time. The upper Protocol Service Layer in TTP/C provides a global clock synchronization between all nodes in the cluster and enables the receiver to detect transmission errors.

Figure 6.2: Basic TTP/C network topologies



Error Information supplied to upper Layer

When a defective frame is received, due to differences in the current control state or corrupted data, notification is sent to the upper layer. Since this information is also sent to the sending node, the sending node can also pass it up to the next protocol layer. Thus the next protocol layer in both nodes can decide how to proceed.

Undetected Errors or not handled by the Layer

This layer can only detect errors resulting from mangled frames, when the communication bus is up an running. Missing frames are not detected and the handling for this errors has to be done in higher layers. Also, frames which have pass the CRC could still contain damaged data. Upper layers have to detect these errors.

6.3.3 Network Layer

TTP/C only supports simple network topologies such as bus and star connections (see figure 6.2). Thus routing is not necessary. Therefore TTP/C does not have any functionality at this level.

6.3.4 Transport Layer

The Protocol Service Layer act as transport layer and provides different groups of services to the higher layers.

Communication Services are responsible for the data exchange, cluster startup and integration, the implicit (and explicit) acknowledgment scheme, and the fault-tolerant clock synchronization. The fault-tolerance aspect is covered by the communication services that operate in all single-fault scenarios.

Safety Services establish the node membership, maintains state consistency among all nodes through a clique avoidance algorithm, controls the independent bus guardian functionality, and maintains active connection among all live nodes through the host controller life sign algorithm. These services guarantees fail-silent behavior of a faulty node in the time domain and prevent the existence of different node groups with unequal controller states.

Higher Level Services are requested by the host and are used for switching between transmission schedules at runtime, synchronizing the cluster with an external clock or an other TTP/C cluster, or for a role-change of a node.

With additional independent hardware units—bus guardians—the bus access is restricted in accordance with the MEDL. In a safety-critical TTP/C configuration, the protocol needs this hardware support to identify and isolated “single fault” errors into so called fault-containment regions (FCR). Many multiple faults can be detected as well. In addition, the protocol specifies higher-level error detection for faults that cannot be tolerated by TTP/C itself.

Several error class can be handled by TTP/C that non fault tolerant protocols can not handle:

an outgoing link failure;
the babbling idiot failure, where one node blocks all others by sends continuously;
masquerading, where a node pretends to be some other node;
slightly-off-specification (SOS), where a node does not send exactly in the correct time slot;
Crash/Omission (CO), where a node fails to send or sends in the wrong time slot; and
massive transient disturbances; where an external impulse, e.g., a strong electromagnetic impulse, disturbs transmission completely.

For further information on the known error classes and their handling in TTP/C consult Kopetz [2, 3]. Though these errors apply to all networks for distributed realtime applications, very few busses are designed to handle them.

Error Detection Capability of the Layer

Error detection takes place in the lower layers and is provided to this layer to be diagnosed.

Error Diagnostic Capability of the Layer

When this layer receives a notification that an error has occurred in the data link layer, the protocol service layer tries to classify the error into one of the several

categories listed above. Error information over several bus cycles is used to make the determination. Since only receiving nodes can decide if a sender's packet is incorrect, the TTP/C protocol uses a membership service to informs every node about the "health-state" of every other node.

Error Correction Capability of the Layer

Once an error is properly diagnosed, the system can take the proper recovery action. Though not all errors can be corrected, secondary errors can be prevented. Usually, this involves isolating the offending node; but in the case of a transient error a reconfiguration or reinitialization may be necessary.

The TTP/C membership service informs every node about the "health-state" of every other node within the theoretical minimal time of one TDMA rounds. A node with an outgoing link failure learns from this membership service that all other nodes assume that it has failed. Once the error is cleared, the node can reestablish consistency autonomously.

Under the assumption of a single fault, the membership protocol diagnosis an omission failure as an incoming or outgoing link failure. After detection with the membership service, the node can reestablish a system wide or only fragmentation wide consistency. For further correction capability, it is up to the application to decide how it should react to an evolving inconsistency.

In a conventional event-triggered system, a faulty node—called babbling idiot—is able to monopolize the communication media by sending high priority messages permanently. In a safety-critical TTP/C configuration, additional bus guardians are used to isolate a babbling idiot failure in an independent fault-containment region. The application itself has to react to this fragmentation.

The TDMA protocol of TTP/C prevents any masquerading fault with a static assignment of slot position (time) to physical node identity. The origin of a message depends only on the time. A global time serves the bus guardians so they can prevent masquerading by only allowing transmissions during the appropriate time slot.

Slightly-off-specification failures—depending on slight differences in time—can occur, but a network structure with guardians reshapes broken frames and eliminates an inconsistency. The guardians guarantees that a node sending out of its time slot can not disturb the correct data transmission of other nodes by blocking the offending node. The sent frame fragment is recognized as error in the data link layer and handled.

The system can not prevent a massive transient failure, but after the disturbance abates the protocol can restart immediately with the time base that has been maintained within the host computer and reestablish a consistent state.

Error handling Information received from upper Layer

TTP/C is a fault-tolerant realtime communication protocol for the implementation of hard realtime applications, “single faults” depending on physical and access protocol errors can be tolerated with only configuration information received from the application layer.

Error Information supplied to upper Layer

While the network itself tries to obtain a consistent network state, this functionality makes no use of application specific knowledge. Any changes in controller state are reported for the upper layer to handle. This is particularly important when a node must be disconnected, so that the overall system integrity can be maintained.

Undetected Errors or not handled by the Layer

In most cases, TTP/C can ensure that node failures do not affect other nodes in the system; however the overall integrity of the system must preserved at the application layer. The network can not decide how to react to a missing node or service. The application must know what other nodes or services can be brought online to compensate the lose; and when the system must reduce functionality or shutdown to prevent further damage to the system.

6.4 Event Channel Network

The **HIDOORS** event service builds on top of a transport layer functionality, that provides the minimum requirement for the network protocol independent session layer. The transport layer must only provide a packet transmission service to package events in byte arrays and to broadcast these packages to the participating nodes. Since this is an extremely light weight protocol, any network protocol realtime guarantee enhance the safety of the application. If the underlaying network protocol only provides a limited packet transmission service, the socket design of the event channel network can be used to bridge this gap thereby providing for a network protocol independent session layer.

6.4.1 Session Layer

The session layer in the event channel network is the foundation for a distributed event service, it creates a session over the network of distributed nodes and implements a multicast for events based on logical channel informations. The session layer can be built on any of the described networks providing a lightweight Java interface for communication in realtime when the underlaying network provides such guarantees.

Error Detection Capability of the Layer

Since this is the first layer of the event channel network, the frame and packet format checks of the lower protocol levels should guarantee diagnostic capabilities on a basic level. Mangled network packages are filtered out by each of the specified communication protocols described above. If the packages received are determined to be correct by the lower layers, the logical channels layer decodes the affiliation of the data to a locally known event channel. Unknown channels are detected here and the application layer is notified. All other errors, when they are not handled transparently for the event channel network session layer, result in a notification or interruption that is thrown with an exception to the application layer.

Error Diagnostic Capability of the Layer

No diagnosis takes place at this level. The system can not distinguish between a bad event channel designator due to a faulty packet and an unknown channel designator due to a registration failure deterministically. A channel lookup could be initialized, but there is in general no upper bound on how long it would take.

Error Correction Capability of the Layer

The session layer of the event channel network provides no correction capability. Each error is passed to the application layer through an exception.

Error handling Information received from upper Layer

The session layer can only know about errors (e.g., missing packets) transmitted from lower protocol layers. This is a platform dependent feature, but the result for the upper presentation layer is transparent. There is no correction capability integrated in the event channel network. Exception are thrown to the application layer to be handled.

Error Information supplied to upper Layer

All errors from the underlying layers and unknown channels errors are passed on.

Undetected Errors or not handled by the Layer

Anything transmission error not caught by the lower layers is not caught here either. All errors both passed up from lower layers and due to unknown channel errors are passed on to the application layer. In the event channel network layer, there is no functionality for a (transparent) retransmission. An exception must be thrown when an error is detected (or reported from a lower layer). While the session layer provides a view of logical channels and events in the event channel network, the

layer itself has no semantic knowledge about this Java objects. If the local node knows the channel and event type id but has no knowledge about what to do with the event, the session layer can not detect the logical error in a communication. This protocol knowledge resides in the presentation layer or even in the application itself.

6.4.2 Presentation Layer

The presentation layer of the event channel network adds components for the event handling to the event channel network. The reception is done quickly and the events are dispatched to registered components to handle them. In addition to errors passed up from below, serialization and deserialization errors can occur.

Error Detection Capability of the Layer

Error detection on this layer relies on the correct use of the provided event transmission, reception, and handling infrastructure. The presentation layer also controls the sequence number of the received and sent events. Duplicate events can be detected and are ignored.

Error Diagnostic Capability of the Layer

This layer provides no further diagnostic capability.

Error Correction Capability of the Layer

The errors notified from lower layers are encapsulated in Java exceptions and passed to an application level handler.

Error handling Information received from upper Layer

The application controls the program logic, like event channel lookup, creation, and announcement. These actions are annotated with time restrictions, and the presentation layer reports any non-compliance. All other information from the application is used for initializing the network configuration and connection.

Error Information supplied to upper Layer

All errors reported from lower layers are forwarded to the application.

Undetected Errors or not handled by the Layer

While the session layer could recognize if a received channel or event type id is registered and known at the local node, it has no ability to enforce the correct use of this informations. The presentation layer also has no knowledge about semantics. Only logical programming errors can be reported to the application layer.

6.5 Application Layer

When using the **HIDOORS** event channel network, overall error handling and system integrity processing must be done in the application. Each error and anomaly of the lower layers is reported to the application. The application can use the high-resolution time functionality of the RTSJ Java extension to set up periodic tasks to monitor communication within a distributed system.

The highest level of errors are application protocol errors, like missing or untimely received events. These errors are undetectable by the event channel network and each (distributed) application has to implement checks and monitor the control flow to guarantee correct time response. To detect and handle errors with communication over the event channel network, one of two conceptually different communication mechanisms (described on page 27) could be used to assure the dependability of the system. For frequent, periodic events, the system can guarantee correct communication flow through event monitoring, otherwise an acknowledge protocol or some other form of feedback must be implemented.

The goal of the lightweight Java API for an event service in realtime is the network protocol and platform independent implementation of the communication functionality in a distributed application. This is achieved by a generic broadcast oriented socket for byte array transmission, implemented on top of the transport layer of the communication network. The remaining differences are discussed in the following sections.

6.5.1 UDP/IP over Ethernet

Even though representative network topologies in a distributed **HIDOORS** application makes no use of network layer functionality with routing, a general implementation based on UDP/IP has to handle possible resulting errors. Because the underlying network protocol provides no reliable packet transmission and the event channel network adds no overall control for lost and duplicate packages this is left to the application layer.

6.5.2 Controller Area Network (CAN)

To provide transmission with CAN within a predefined time limit, a selection of opportune identifiers is necessary. CAN give better realtime behavior than UDP/IP,

but it is still not applicable for hard realtime systems. Explicit network duplication must also be used to provide fault tolerance.

6.5.3 Time-Triggered Protocol, SAE class C (TTP/C)

TTP/C is designed to provide hard realtime behavior. Its fault tolerant architecture removes many errors transparently for the upper layers. Behavior can be improved further by using bus guardians to separate faulty nodes in the communication bus to guarantee consistency when failures occur. Still, the application itself must respond to node failure to insure overall system integrity.

6.6 Summary

The event service is designed as a light weight service for embedded systems. There is very little error handling in the service itself. This means that the application needs to monitor message traffic to ensure the system is running correctly. Safety-critical application should use a realtime fault tolerant network to insure system integrity. The event service is well suited for other applications like providing user monitoring of a safety-critical system from non safety application. Here, standard network protocol can be used since the event service decouples the sender from the receiver. The service is only as reliable as its underlying network.

Chapter 7

Testing

For testing purpose, the packages and applications in **`org.hidoors.test`** are provided. This package contains several different simple test applications. They are all based on the **`UDPBroadcastSocket`** implementation with different channels. These examples test the whole functionality of the the event channel network. Beginning with the tests of basic features the examples in section 7.1 extends the test environment until dynamic features. Each test description starts with a short list of the main issues shown with this test application. The section 7.2 describes the common elements of all application that use the event service. Because of the uniform structure section 7.3 introduces the implementation with a XML/XSLT based transformation technique based on a declarative description presented in chapter 5.

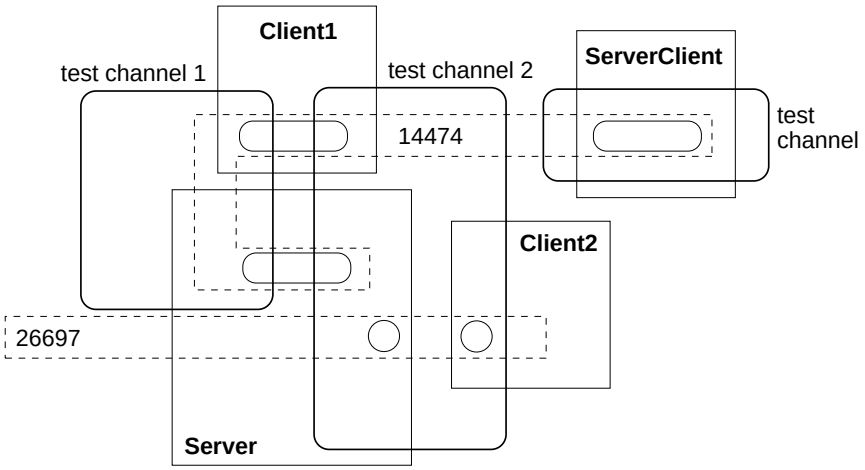
7.1 Test programs

The test programs are integrated in the event channel network API and make use of a DEBUG mechanism to test the infrastructure. For all examples, the ant configuration file **`build.xml`** provider special targets. To use ant with **Jamaica**, the **`bootstrap.xml`** has to be executed first (**`ant -f bootstrap.xml`**).

The JVM environment parameter **`hidoors.debug`** sets the DEBUG level of each program and the library classes. The class **`org.hidoors.DEBUG`** defines for each class a debugging code and a method to print debug information if the bits for the calling class are set with the level. All debug output is disabled if you use the standard **`build.xml`**, with the **JamaicaVM** targets!

If one starts the testing programs manually, one can set this parameter. More information is provided in the following subsections detailing each of the examples. There is often a **`jamaicavm.*`** target, for interpreted execution. To build binaries, one uses the tasks with **`build.*`**. Targets with a **`.db`** ending produce binaries with maximum debug information. The **JamaicaVM** does not allow VM arguments.

Figure 7.1: Basic examples



7.1.1 Simple Server and Clients

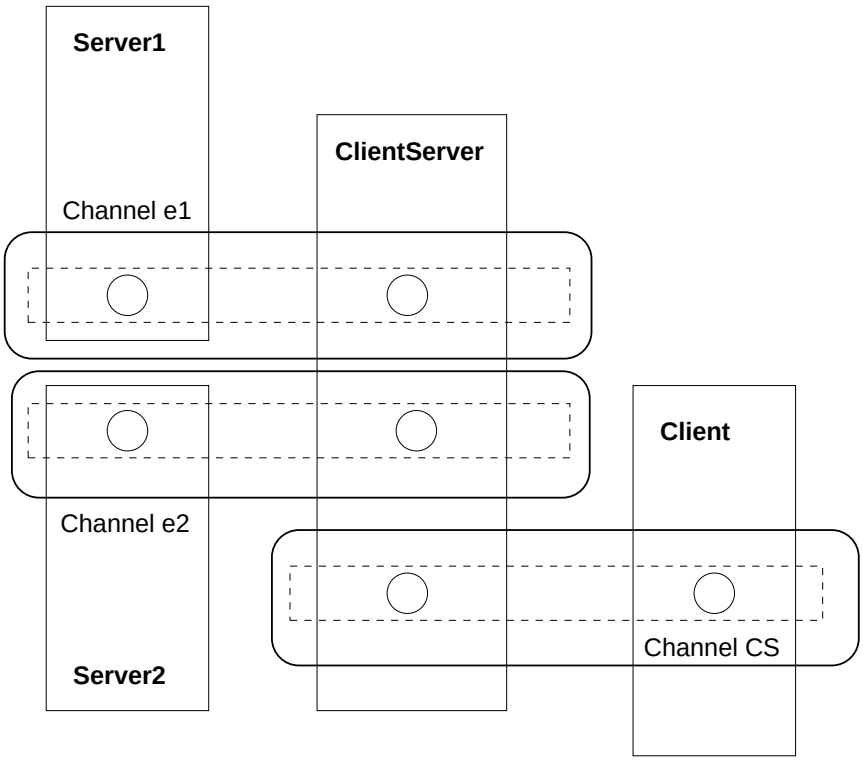
The following issues are addressed in this example:

channel creation and lookup,
communication over different channels, and
server/client structure of distributed application.

The test environment, with some basic examples in **`org.hidoors.test.services.event`**, is illustrated in figure 7.1. The sample applications uses some basic channels and sockets to demonstrate the event communication. It is a one-sided communication and the receiving clients starts simple actions to print out the events.

`Server` creates two test channels and uses the class **`ServerAction`** to send events in a loop over two UDP sockets (**`UDPBroadcastSocket`**s at port 14474 and 26697). The first channel only uses port 14474 and the second channel receive and transmit over both ports.
`Client1`queries for both test channels and registers two handlers implemented as inner class Runnables with this channels. The dispatched events are passed to two different actions (**`PrintAction`**, **`ClientAction`**).
`Client2`only queries for the second channel and registers a handler to start different actions (**`PrintAction`**, **`ClientAction`**) for the two known events types.
`ServerClient` is a server-client-node. It starts a server with one channel to transmit and a client to receive this events over the network but in the same VM.

Figure 7.2: Collecting ClientServer node



Each program can use the JVM parameter **`hidoors.debug`** to set the DEBUG level, which is described in the class **`org.hidoors.DEBUG`** and is used to control the DEBUG output of the API classes.

Use **`ant jamaicavm.*`** or **`ant build.*`** with **`server`**, **`client1`**, **`client2`**, and **`serverclient`** to start this examples with DEBUG output.

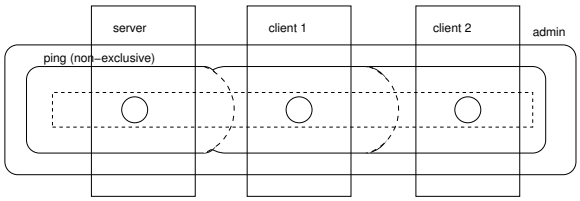
7.1.2 Different Servers a ClientServer and a Client

The following issues are addressed in this example:

channel creation and lookup,
communication over different channels,
server/client structure of distributed application, and
collecting/filtering of events.

This example shows the same functionality as the examples in section 7.1.1. The example builds a typically used pattern with a barrier to act as collecting proxy for the real client.

Figure 7.3: Ping Server and Client



The environment in figure 7.2 shows two servers, sending events over two different channels with different communication networks. A special client-server collects these events and resends the information to a connected other client. These classes are located in the package **`org.hidoors.test.eka`** and work with the same standard mechanisms. The use of the necessary infrastructure is described in section 7.2.

This program example is also a prototype for the connection of realtime bus systems to non-realtime infrastructures. The pure client could be an application in Java standard edition, for example a monitor system, without the need of realtime access. For the connection of such 'normal' Java the API will provide an adapter with less reliability. The guarantees of the realtime network and the error handling and failure model described in chapter 6 is obsolete there.

7.1.3 Ping Server and Client

The following issues are addressed in this example:

channel creation and lookup,
two-way communication over a channel,
server/client structure of distributed application, and
application protocol with request-response.

This client-server example is a exemplar for a simple two-way communication. The figure 7.3 shows one channel with a connected server. This server creates a non-exclusive channel with the name 'ping'. First this channel only include the server node. The two also shown clients search for this 'ping' channel and participate by time.

When a client is included in the channel it could send a PING event over the channel and for any received PING event the server react with some answer events.

There is no connection oriented behavior in this example, each client node receives all events. To implement nodes using direct connections while communicating the application layer using the event channel network has to implement such a protocol. The similar example described in the next section add this and additional control with timeouts.

7.1.4 Timeout Example

The following issues are addressed in this example:

channel creation and lookup,
two-way communication over a channel,
server/client structure of distributed application, and
application protocol with regular events, sent frequently

The example in **`org.hidoors.test.timeout`** has a structure similar to figure 7.3. in this example a server is controlled by one client node. If an event timeout is exceeded the client sends a RESTART/RESET event to the server. The server receives the control event and is the client is authorized to control it tries to reset the sending component. The server implementation uses the RTSJ high-resolution time API to send regular events and when an internal counter exceed a borderline a defect is simulated, the server could forget to send the event. The authorized client has to reset the server again.

This example shows the need for an application level error handling and connection oriented protocol layer. The example uses simple identifiers but cryptographic mechanisms, like SSL, could be used too. The event channel network itself gives no further guarantees for a reliable network (see chapter 6), even the existing features of the underlaying bus system will be lost.

7.1.5 Dynamic Generation

The following issues are addressed in this example:

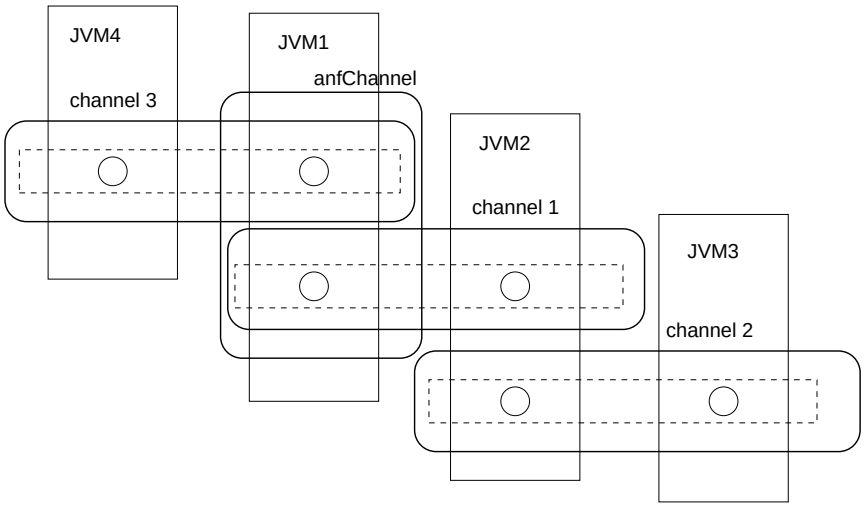
channel creation and lookup,
communication over different channels,
server/client structure of distributed application, and
channel and server/client creation on demand.

This section shows further dynamic features of the event channel API. The predefined infrastructure (compare section 7.2.1) could be enhanced with new sockets and new channels too. Because of the new priorities and perhaps new capacities needed each parameter of an exiting receiver infrastructure is designed fully dynamic.

By calling methods in **`EventChannelFactory`** and **`PushEventReceiver`** the dynamic behavior is enabled. One example (depicted in figure) is **`org.hidoors.test.eka.dynamisch2`** with four different VMs and three physical UDP/IP broadcast networks.

JVM1 uses two UDPBroadcastSockets and creates dynamically a new server to send events to JVM2. The first server sends this information to the created client/server in JVM1.

Figure 7.4: Dynamic example with 4 VMs



JVM2 receives the events (client part) and sends (server part) them to JVM3 over a new channel (socket address and name got from JVM1).
JVM3 searches for any channel and echos that events.
JVM4 (would be better called -1) connects to another channel created by JVM1.

7.2 Application Implementation

All application that use the event service share some common elements. The structure of the test examples are shown in subsection 7.2.1 and can be used as a reference. Each application is divided in infrastructure and logic parts (application layer).

The infrastructure needed on each node is always the same.

1. The receiver (and transmitter) must be configured with capacity and priorities for the event queues. These parameters could be dynamically changed (see section 7.1.5) but the hard realtime aspects verified with a static realtime analyzer could not check this in advance.

2. All known initial sockets are created, dynamic creation limits the static verification.

3. The **`EventChannelFactory`** is then created with the initial sockets by starting the receiver for the administration channel on each socket. This functionality can also be extended by method calls later.

After the infrastructure is initialized, the logic parts can create and use new channels, search for channels by symbol, receive announcements of new channels, and send events or handle the events received over the channels. A component design structure can be facilitated by implementing application logic in **`AbstracAction`** subclasses, thus separating it from infrastructure code. The look up mechanism of a client application should be implemented in a separate thread to prevent dead locks with its own server component.

7.2.1 Structure of the programs

The first part in each application is the creation of the receiver-/transmitter-infrastructure (compare chapter 4).

> Create a **`PushEventReceiver`** (and/or **`PushEventTransmitter`**).
> The receiver is created first because the used data structure for **`Activity`** threads and the queues for each event priority is configured there. To enable a fully generic behavior of all nodes with lookup and creation of new sockets and channels the receiver provides methods to change this parameters in a running program too. For an example see section 7.1.5. If the application is a pure server a standard receiver is created when creating the transmitter singleton!
> Create an **`EventSocket`** for each used **`BroadcastSocket`**.
> Create **`EventChannelFactory`** with these **`EventSockets`**.
> For further changes the the factory provides the method **`addAdminSocket(EventSocket)`**.

After that, a server uses the factory to create new channels (**`createChannel(parameter)`**) and send events.

A client tries to find a specific channel in his local factory cache (**`getChannel(symbol)`**) and uses the administration channel system wide lookup mechanism (**`getChannels(timeout)`**) to QUERY, compare section 4.4.

This simple control flow is always the same and could easily be generated. This feature of the event channel network is described in the next section 7.3.

7.3 Implementation of XSLT based applications

Since the structure of each application in the event channel network is always the same, it is easy to generate the executables out of an XML file based declarative description. A DTD for this is given in the appendix. The class **`XSLTNodeFactory`** transforms a given XML description with an XSLT file **`node.xslt`** into a Java application with server and/or client functionality for one node. The temporary generated source code is compiled with the

CodeGeneration class. For more information on the style sheet see section B.3.

The generated application uses action objects of **AbstractAction** subclasses to implement application functionality. The likewise generated **PushEventHandler** is also declared in the XML file. The binding of action objects to events received is defined in the file as well.

Each action is a subclass of **AbstractAction**. These objects should be created with a factory method for better control (action could be a singleton): **getAction()**. The defined abstract template method **doAction()** is used to implement the action.

The main functionality of a server is encapsulated in a action object too and the application is started with that **doAction()**.

7.3.1 XML and XSL Transformation Example

The following issues are addressed in this example:

channel creation and lookup,
communication over different channels,
server/client structure of distributed application, and
description of server and client functionality with XML.

The examples testing the XSL transformation feature (some classes and XML files in **org.hidoors.test.services.event**) of the API are started with **antxslt.server|xslt.client**.

By starting the classes **XSLTServer** and **XSLTClient** that will create a **XSLTNodeFactory**.

The files **server.xml**, **client1.xml**, and **client2.xml** are used to create and start executables with the same functionality as the pure Java applications described above in section 7.1.1 and 7.1.2. However they must first be transformed into an application.

Chapter 8

Conclusion

The event service protocol is designed for transmitting events between nodes in a realtime, embedded system where decoupling of sender and receiver is necessary for dynamic reconfiguration. Since building fault tolerant systems is one likely application, there can be no central point of failure and thus there is no central event manager. The service provides an easy to implement infrastructure for implicit invocation communication in distributed applications with demands on realtime. It provides a lightweight, event channel supported Java communication service API with a deterministic behavior. A basic prototype has been implemented and tested to demonstrate its feasibility. A full evaluation awaits more extensive applications.

Appendix A

Structure of the API

This appendix gives a complete overview of the classes and interfaces provided in the Java service API. It is structured by means of the Java packages in **`org.hidoors`**:

Main package: **`org.hidoors.services.event`**

`EventChannel,EventChannelFactory`
Logical channel and creation factory see chapter 3
`BroadcastSocket`
Socket to the communication network to send/receive byte arrays

`UDPBroadcastSocket,UDPBroadcastCollectionSocket`
Example implementations based on UDP/IP see chapter 2
`CANBroadcastSocket`
Example implementations based on CAN see chapter 2
`EventSocket`
Wrapper to extend the simple byte array broadcast socket to handle event objects in channels.
`RemoteEvent`
Generic object (event) for communication between nodes.
`PushEventTransmitter,PushEventReceiver`
Singletons as connection between Java applications and the physical sockets, with ***`PushEventReceiver.SocketReceiver`*** *threads listening for events on each socket* see chapter 4
`ActivityManager`
Handles incoming events by assigning an event to a thread from a pool of thread of the class ***`ActivityManager.Activity`*** see chapter 4
`PushEventHandler`
Application provided logic (implements java.lang.Runnable) to be executed with an activity thread see chapter 4
`AdminChannel,AdminChannelHandler`

Protocol for a predefined administration channel with handler logic for the channel creation phase in a network see section 4.4

`AbstractAction`

Root class for actions used through dynamically created applications described with XSLT see section 7.3

associated exceptions, and some classes to support the channel creation protocol and event serialization

Support package: **`org.hidoors.util`**

`Queue,BlockingQueue,BlockingPriorityQueue`

Java data structures implementing (non-)blocking FIFO queues, and associated exceptions. see chapter 5

`CodeGeneration`

Utility class for dynamic class generation, and compilation in a temporary directory structure see section 7.3

XSLT support package: **`org.hidoors.util.xslt`**

`XSLTransformationFactory`

Factory to create XSL transformer with given XSLT file to support dynamic program generation see section B.3

Test packages: ***`org.hidoors.test`***

Examples to test the Event Channel Network see section 7.2 and 7.3

Appendix B

Diagrams and XML/XSLT

B.1 Main classes and associations

The main classes of the whole framework and their relationship are shown in figure B.1. For a better overview the Exception classes and the associations to the utility classes are not shown.

B.2 DTD for the description of a node

This section shows the DTD for XML descriptions and XSLT file used in section 7.3. The DTD is also depicted as UML diagram in figure B.2.

The root element is called "nodes". It contains client and server node declarations and the sockets collection.

```
<!ELEMENT nodes (socket+, channel+, server+, client+)>
```

A socket only has a an id and a code attribute. The code line is used to instantiate a channel.

```
<!ELEMENT socket EMPTY>
<!ATTLIST socket
id ID #REQUIRED
code CDATA #REQUIRED
>
```

A channel has a name (string), a priority (int), is exclusive or not, has a minimal capacity (int), and its queued elements are overwritable or not. It spans at least one socket and transports at least one event type over it.

```
<!ELEMENT channel (socket-ref+, event-type+)>
<!ATTLIST channel
```

Figure B.1: Main classes and associations

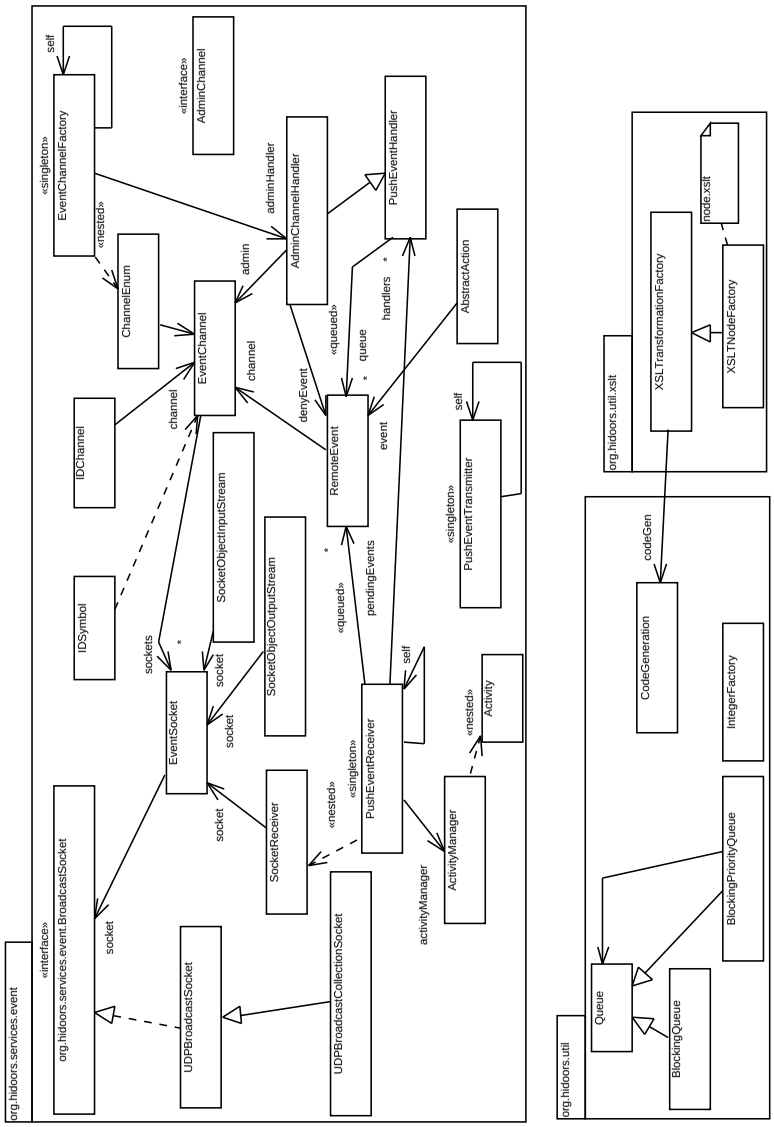
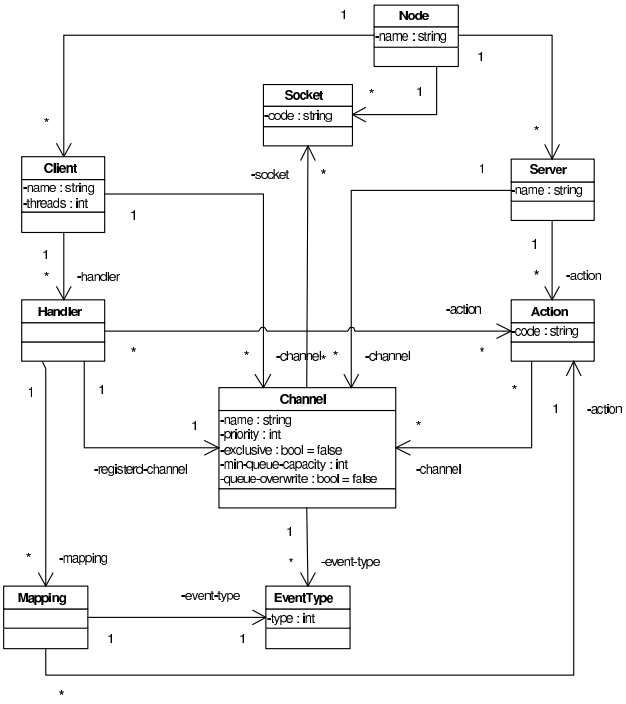


Figure B.2: DTD entities shown in UML diagram



```
name CDATA #REQUIRED
priority CDATA #REQUIRED
exclusive (true|false) "false"
min-queue-capacity CDATA #REQUIRED
queue-overwrite (true|false) "false"
>
```

This element simply references a socket.

```
<!ELEMENT socket-ref EMPTY>
<!ATTLIST socket-ref
ref IDREF #REQUIRED
>
```

An event type only has an id and a type attribute (int).

```
<!ELEMENT event-type EMPTY>
<!ATTLIST event-type
id ID #REQUIRED
type CDATA #REQUIRED
>
```

A client node has a name and a number of threads (int). It contains at leas one handler and one action declaration in arbitrary order.

```
<!ELEMENT client (action+, handler+)>
<!ATTLIST client
name ID #REQUIRED
threads CDATA #REQUIRED
>
```

A server node has a name attribute. It contains at least one action declaration and one reference on the channels used by this server.

```
<!ELEMENT server (channel-ref+, action+)>
<!ATTLIST server
name ID #REQUIRED
>
```

This element simply references a channel.

```
<!ELEMENT channel-ref EMPTY>
<!ATTLIST channel-ref
ref IDREF #REQUIRED
>
```

A client node has a name and a number of threads (int). It contains at least one handler and one action declaration.

```
<!ELEMENT client (handler+, action+)>
<!ATTLIST client
name ID #REQUIRED
threads CDATA #REQUIRED
>
```

A handler maps an action to each event type on a certain channel.

```
<!ELEMENT handler (mapping+)>
<!ATTLIST handler
channel IDREF #REQUIRED
>
```

A mapping maps one certain event type on one certain action. The event type has to belong to the handlers channel, the referenced action has to be one of the handlers actions.

```
<!ELEMENT mapping EMPTY>
<!ATTLIST mapping
event-type IDREF #REQUIRED
action IDREF #REQUIRED
>
```

An action simply encapsulates a line of code.

```
<!ELEMENT action EMPTY>
<!ATTLIST action
id ID #REQUIRED
code CDATA #REQUIRED
>
```

B.3 XSLT file for simple client/server nodes

This XSLT script is used to convert the example XML files in the next section to generate the applications shown in section 7.1.

The following XSL style sheet describes the transformation of a XML file to Java code used to generate simple applications:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:java="http://xml.apache.org/xslt/java"
   exclude-result-prefixes="java">
  <xsl:output method="text" />
```

Create Java code for the root element node (server or client):

```
<xsl:template match="/">
    <xsl:apply-templates select="/nodes" />
  </xsl:template>
  <xsl:template match="nodes">
```

Create the necessary imports:

```
<xsl:text disable-output-escaping="yes">
  import org.hidoors.DEBUG;
  import org.hidoors.services.event.PushEventTransmitter;
  import org.hidoors.services.event.PushEventReceiver;
  import org.hidoors.services.event.PushEventHandler;
  import org.hidoors.services.event.BroadcastSocket;
  import org.hidoors.services.event.RemoteEvent;
  import org.hidoors.services.event.EventSocket;
  import org.hidoors.services.event.EventChannelFactory;
  import org.hidoors.services.event.EventChannel;
  import org.hidoors.services.event.AbstractAction;
  import org.hidoors.services.event.ChannelException;
  import java.util.Enumeration;
```

Read class name, create class code and standard constructor:

```
/** @author generated code! */
  public class </xsl:text>
    <xsl:value-of select="@name"/>
    <xsl:text disable-output-escaping="yes"> {
  /** standard constructor */
  public </xsl:text><xsl:value-of select="@name"/>
    <xsl:text disable-output-escaping="yes">() {
```

Create client infrastructure code if defined in XML file:

```
<xsl:if test="not(not(client))">
    <xsl:call-template name="client-infrastructure">
    </xsl:call-template>
  </xsl:if>
```

Create server infrastructure code if no client defined in XML file:

```
<xsl:if test="not(not(server))">
    <xsl:call-template name="server-infrastructure">
    </xsl:call-template>
  </xsl:if>
```

Create code for local sockets to the communication networks:

```
<xsl:call-template name="sockets">
  </xsl:call-template>
```

Create code only for clients:

```
<xsl:if test="not(not(client))">
    <xsl:call-template name="client">
    </xsl:call-template>
  </xsl:if>
```

Create code only for servers:

```
<xsl:if test="not(not(server))">
    <xsl:call-template name="server">
    </xsl:call-template>
  </xsl:if>
```

Declare member variables for clients:

```
<xsl:if test="not(not(client))">
    <xsl:call-template name="client-var">
    </xsl:call-template>
  </xsl:if>
  <xsl:text disable-output-escaping="yes">
  } // </xsl:text><xsl:value-of select="@name"/>
  <xsl:text disable-output-escaping="yes">
  </xsl:text></xsl:template>
```

The following XSLT code is called as templates from the above declared structure.
Socket template: Create sockets, create EventChannelFactory:

```
<xsl:template name="sockets">
    <xsl:for-each select="socket">
      <xsl:value-of select="concat('EventSocket ', generate-id(.),
        ' = new EventSocket(new ', ./class/text(), '(')" />
      <xsl:for-each select="param">
      <xsl:value-of select="./text()"/>
      <xsl:if test="position()!=last()">
        <xsl:value-of select="', '"/>
      </xsl:if>
      </xsl:for-each><!-- param -->
    <xsl:text disable-output-escaping="yes">));</xsl:text>
```

```
      </xsl:for-each><!-- socket -->
      <xsl:text disable-output-escaping="yes">
      EventSocket[] esa = new EventSocket[] {
      </xsl:text>
      <xsl:for-each select="socket">
        <xsl:value-of select="generate-id(.)"/>
        <xsl:if test="position()!=last()">
          <xsl:value-of select="', '"/>
        </xsl:if>
      </xsl:for-each><!-- socket -->
      <xsl:text disable-output-escaping="yes"> };
        EventChannelFactory factory =
          EventChannelFactory.getFactory(esa);
      </xsl:text>
    </xsl:template><!-- sockets -->
```

Server infrastructure template:

```
<xsl:template name="server-infrastructure">
```

Create server code to start transmitter:

```
<xsl:text disable-output-escaping="yes">
  // create Transmitter-Handler-Infrastructure
  PushEventTransmitter transmitter =  PushEventTransmitter.
    getPushEventTransmitter();
  </xsl:text>
  </xsl:template><!-- server-infrastructure -->
```

Create code for channel creation for referenced sockets:

```
<xsl:template name="server">
  <xsl:for-each select="server/channel">
  <xsl:text disable-output-escaping="yes">{
    EventChannel channel = null;
    try {
    channel = factory.createChannel("</xsl:text>
      <xsl:value-of select="concat(@name, '&amp;quot;, ',
      ./priority/text(), ', ')" />
```

... with parameter list:

```
<xsl:choose>
  <xsl:when test="string(exclusive)='true'">
```

```
    <xsl:text disable-output-escaping="yes">true</xsl:text>
  </xsl:when>
  <xsl:otherwise>
    <xsl:text disable-output-escaping="yes">false</xsl:text>
  </xsl:otherwise>
  </xsl:choose>
  <xsl:value-of select="', new EventSocket[] {'" />
  <xsl:for-each select="socket-ref">
    <xsl:value-of select="generate-id(../../../socket
                          [@name=current()/text()])" />
    <xsl:if test="position()!=last()">
      <xsl:value-of select="','" />
    </xsl:if>
  </xsl:for-each><!-- socket-ref -->
    <xsl:text disable-output-escaping="yes">});
    } catch (ChannelException ce) {
      ce.printStackTrace();
      System.exit(0);
    }
  }</xsl:text>
  </xsl:for-each><!-- server/channel -->
```

Server actions, create code one thread for each defined action class, created with factory method and started:

```
<xsl:for-each select="server/action">
  new Thread(<xsl:value-of select="./class/text()" />
  <xsl:text disable-output-escaping="yes">
  .getAction(new Object [] {</xsl:text>
             <xsl:for-each select="param">
             <xsl:value-of select="./text()"/>
             <xsl:if test="position()!=last()">
               <xsl:value-of select="', '"/>
             </xsl:if>
             </xsl:for-each><!-- param -->
  <xsl:text disable-output-escaping="yes">})).start();
  </xsl:text>
  </xsl:for-each><!-- server/action -->
```

... generate code to close constructor for server node:

```
<xsl:text disable-output-escaping="yes">
  } // constructor
  </xsl:text>
  </xsl:template><!-- server -->
```

Client infrastructure template:

```
<xsl:template name="client-infrastructure">
```

Create code to start receiver with queues for expected events:

```
<xsl:text disable-output-escaping="yes">
  // create Receiver-/Handler-Infrastructure
  PushEventReceiver receiver =  PushEventReceiver.
    getPushEventReceiver(new String[] {</xsl:text>
  <xsl:for-each select="client/queue">
    <xsl:value-of select="description/text()"/>
    <xsl:if test="position()!=last()">
      <xsl:value-of select="', '" />
    </xsl:if>
  </xsl:for-each><!-- client/queue -->
  <xsl:text disable-output-escaping="yes">}, new int[] {
  <xsl:for-each select="client/queue">
    <xsl:value-of select="priority/text()"/>
    <xsl:if test="position()!=last()">
      <xsl:value-of select="', '" />
    </xsl:if>
  </xsl:for-each><!-- client/queue -->
  <xsl:text disable-output-escaping="yes">}, new int[] {
  </xsl:text>
  <xsl:for-each select="client/queue">
    <xsl:value-of select="capacity/text()"/>
    <xsl:if test="position()!=last()">
      <xsl:value-of select="', '" />
    </xsl:if>
  </xsl:for-each><!-- client/queue -->
  <xsl:text disable-output-escaping="yes">}, new boolean [] {
  </xsl:text>
  <xsl:for-each select="client/queue">
  <xsl:choose>
  <xsl:when test="string(overwrite)='true'">
    <xsl:text disable-output-escaping="yes">true</xsl:text>
  </xsl:when>
  <xsl:otherwise>
    <xsl:text disable-output-escaping="yes">false</xsl:text>
  </xsl:otherwise>
  </xsl:choose>
  <xsl:if test="position()!=last()">
    <xsl:value-of select="', '" />
```

```
  </xsl:if>
  </xsl:for-each><!-- client/queue -->
  <xsl:text disable-output-escaping="yes">}, </xsl:text>
  <xsl:value-of select="client/activities/text()" />
  <xsl:text disable-output-escaping="yes">);</xsl:text>
  </xsl:template><!-- client-infrastructure -->
```

Lookup for described channels:

```
<xsl:template name="client">
  <xsl:for-each select="client/handler">
  <xsl:for-each select="channel-ref">
  <xsl:text disable-output-escaping="yes">do {
    Enumeration cachedChannels = factory.getChannels(5000);
    while (cachedChannels.hasMoreElements()) {
      EventChannel channel =
        (EventChannel) cachedChannels.nextElement();
      if (channel.equalsSymbol("</xsl:text>
      <xsl:value-of select="." />
      <xsl:text disable-output-escaping="yes">")) {</xsl:text>
        <xsl:value-of select="concat('_',
          generate-id(../../channel[@name=current()/text()]))"/>
        <xsl:text disable-output-escaping="yes"> = channel;
        break;
      }
    }
  } while (</xsl:text>
  <xsl:value-of select="concat('_',
    generate-id(../../channel[@name=current()/text()]))"/>
  <xsl:text disable-output-escaping="yes"> == null);
  </xsl:text>
  </xsl:for-each><!-- channel-ref -->
  <xsl:value-of select="concat('_', generate-id(.), ' =
    new PushEventHandler(20, new ')" />
  <xsl:if test="not(logic)">
  <xsl:text disable-output-escaping="yes">Runnable() {
  public void run() {
  </xsl:text>
  <xsl:value-of select="concat('RemoteEvent event =
    _', generate-id(.), '.getEvent();')" />
  <xsl:value-of select="concat('switch (event.getType()) {')" />
    <xsl:for-each select="event-type">
    <xsl:value-of select="concat('case ', @name, ':')" />
      <xsl:for-each select="action">
```

```
          <xsl:value-of select="concat('_',
            generate-id(.), '.setEvent(event);')"/>
          <xsl:value-of select="concat('_',
            generate-id(.), '.run();')"/>
        </xsl:for-each><!-- action -->
      <xsl:text disable-output-escaping="yes">break;</xsl:text>
      </xsl:for-each><!-- event-type -->
      <xsl:text disable-output-escaping="yes">
      default:
        System.out.println(" Client1, unknown: " + event);
        break;
      }
    }
    });</xsl:text>
    </xsl:if><!-- logic -->
    <xsl:if test="not(not(logic))">
      <xsl:value-of select="logic/text()" />
    </xsl:if>
    <xsl:value-of select="concat('receiver.addHandler(_',
      generate-id(../channel[@name=current()/channel-ref/text()]), ',
      _', generate-id(.), ');')" />
    </xsl:for-each><!-- client/handler -->
    <xsl:text disable-output-escaping="yes">
    } // constructor
    </xsl:text>
    </xsl:template><!-- client -->
```

Client variables template:

```
<xsl:template name="client-var">
    <xsl:for-each select="client/channel">
    <xsl:value-of select="concat('EventChannel _',
      generate-id(.), ';')" />
    <xsl:for-each select="event-type">
      <xsl:value-of select="concat('public static final int ',
        @name, ' = ', ./type/text(), ';')" />
    </xsl:for-each><!-- event-type -->
  </xsl:for-each><!-- client/channel -->
```

Client handlers template, create dispatching and add register with PushEventReceiver:

```
<xsl:for-each select="client/handler">
    <xsl:value-of select="concat('PushEventHandler _',
```

```
        generate-id(.), ';')" />
      <xsl:for-each select="event-type">
      <xsl:for-each select="action">
        <xsl:value-of select="concat('AbstractAction _', generate-id(.),
          ' = ', ./class/text(), '.getAction(new Object[] {')" />
        <xsl:for-each select="param">
          <xsl:value-of select="./text()"/>
          <xsl:if test="position()!=last()">
            <xsl:value-of select="', '"/>
          </xsl:if>
        </xsl:for-each><!-- param -->
      </xsl:for-each><!-- action -->
    </xsl:for-each><!-- event-type -->
  </xsl:for-each><!-- client/handler -->
  </xsl:template><!-- client-var -->
  </xsl:stylesheet>
```

B.4 Example XML files

XML files for the test applications shown in section 7.1.

Simple Server example, name of the generated class:

```
<node name="Server">
```

List of local known sockets:

```
<socket name="udp14474">
  <class>org.hidoors.services.event.
           UDPBroadcastSocket</class>
  <param>14474</param>
</socket>
<socket name="udp26697">
  <class>org.hidoors.services.event.
           UDPBroadcastSocket</class>
  <param>26697</param>
</socket>
```

Configuration for server: Logical channels with known event types and associated with the sockets bey name:

```
<server>
  <channel name="test channel 1">
    <exclusive>true</exclusive>
    <event-type name="TEST1_TYPE">
      <type>35</type>
```

```
        <class>java.lang.String</class>
      </event-type>
      <priority>2</priority>
      <socket-ref>udp14474</socket-ref>
    </channel>
    <channel name="test channel 2">
      <event-type name="TEST1_TYPE">
        <type>33</type>
        <class>java.lang.String</class>
      </event-type>
      <event-type name="TEST2_TYPE">
        <type>34</type>
        <class>java.lang.String</class>
      </event-type>
      <priority>1</priority>
      <socket-ref>udp14474</socket-ref>
      <socket-ref>udp26697</socket-ref>
    </channel>
```

The server executes the defined action, a subclass of AbstractAction (object creation with factory method!):

```
    <action>
      <class>org.hidoors.test.services.event.
               ServerAction</class>
      <param>new Integer(10000)</param>
    </action>
  </server>
  </node>
```

The XML description for a client or a server/client node is similar.

Bibliography

[1] Robert Bosch GmbH. *CAN Specification*. Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, 2.0 edition, September 1991.

[2] H. Kopetz. A Comparison of TTP/C and FlexRay. Technical report, Technical University of Vienna, May 2001.

[3] H. Kopetz. Fault Containment and Error Detection in TTP/C and FlexRay. Technical Report 1.5, Technical University of Vienna, August 2002.

[4] R. Obermaisser. CAN Emulation i a Time-Triggered Environment. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)*, 2002.

[5] Object Management Group. *Event Service Specification*, 1.1 edition, March 2001.

[6] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition, September 2001.

[7] Object Management Group. *Real-Time CORBA Specification*, 1.1 edition, August 2002.

[8] J. Postel. Internet Protocol. RFC 760, January 1980.

[9] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[10] Dirk Riehle. The Event Notification Pattern – Integrating Implicit Invocation with Object-Orientation. *Theory and Practice of Object Systems 2*, 2(1), 1996.

[11] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall Inc., 1989.

[12] William Stallings. *Data and Computer Communications*. Prentice-Hall, Inc., 5th edition, 1996.

[13] Sun Microsystems. *InfoBus 1.2 Specification*, February 2004.

[14] Ken Tindell and Alan Burns. Guaranteeing Message Latencies on Control Area Netwrk (CAN), 1994.

[15] TTA-Group, Schoenbrunner Strasse 7, A-1040 Vienna, Austira. *Time-Triggered Protocol TTP/C. High-Level Specification Document*, 1.0.0 edition, July 2002.

[16] Zhi Wang, Ye_qiong Song, Ji_ming Chen, and You_xian Sun. Real Time Characteristics of Ethernet and Its Improvement. In *Proceedings of the* 4th *World Congress on Intelligent Control and Automation*, June 2002.